Large Language Models: Building General Coding Assistants

Auteur : Daoud, Samuel
Promoteur(s) : Ernst, Damien
Faculté : Faculté des Sciences appliquées
Diplôme : Master : ingénieur civil en science des données, à finalité spécialisée
Année académique : 2023-2024
URI/URL : http://hdl.handle.net/2268.2/20996

University of Liège
Faculty of Applied Sciences

Large Language Models: Building General Coding Assistants

A dissertation submitted in partial fulfillment of the requirements for the degree of
Master of Science in Data Science and Engineering

Author
Samuel Daoud

Advisor
Professor Damien Ernst

2023-2024

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my supervisor, Prof. Damien Ernst, for his invaluable advice, patience, and encouragement throughout the course of my research. I am particularly grateful for the opportunity to apply this research in an industry context, gaining valuable insights into the deployment of AI in the domain.

I would also like to thank Lize Pirenne for her precious support, constructive feedback and suggestions that greatly contributed to the improvement of this work. She was always willing to help whenever I needed assistance.

My sincere thanks also go to Philippe Drugmand for the trust and guidance during my time spent at Haulogy.

To my dear friends, your cheerfulness and positivity made this journey more than enjoyable.

Lastly, I would like to thank my family for their support and encouragement. To my parents, thank you for providing me with the best environment to pursue my studies. Thank you for always believing in me, your patience and love kept me going even during the most challenging times.

To everyone mentioned and many more who supported me along the way, thank you.

Abstract

In the realm of software development, the frequent release of new Application Programming Interface (API) versions presents a significant challenge for engineers and developers. Traditionally, adapting to these changes requires a comprehensive update of the entire application, resulting in considerable time and resource investments. This situation highlights the need to support developers in managing the numerous tedious tasks they encounter daily.

This thesis addresses these challenges by leveraging Large Language Models (LLMs) for code-related tasks and introduces a framework for deploying advanced general coding assistants that achieve state-of-the-art performance.

The approach involves selecting and deploying a model based on several meaningful criteria, choosing appropriate benchmarks and datasets for fine-tuning, and developing a framework capable of fine-tuning on a single GPU. We also deploy our own benchmark, building upon the dataset released in previous related works.

We address the limitations associated with fine-tuning under constrained computational resources. Our fine-tuned models demonstrate a systematic improvement in performance for the specific downstream tasks they are adapted to. Improving their precision up to 206.25%.

We also provide critical insights into both the evaluation metrics for LLMs and the limitations of current benchmarks.

Contents

Chapter 1

Introduction

1.1 Context

Figure 1.1: Haulogy

This research was conducted at Haulogy.

Haulogy is a company that provides modular and scalable management software and consulting services to businesses in the energy and utilities sector. They offer solutions and services for various markets such as distribution, supply, e-mobility, flexibility, and energy sharing, targeting different players like DSO's, retailers, B2B consumers, BRP's, aggregators, and e-mobility service providers. Haulogy has a presence in three European countries and focuses on building long-term relationships with their clients. Their expertise includes artificial intelligence applications in the energy sector, particularly in new models of energy provision, highlighting the growing importance of AI in the industry.

As an intern, I worked in the R&D team to explore how modern AI models could provide solutions for the most important challenges the firm is currently facing.

1.2 Identifying Challenges

During my internship at Haulogy, I spent my time spotting and discussing the different challenges currently faced by the firm. From the beginning, the objective was to explore the possibility to engineer and deploy modern AI tools, particularly smart chatbot assistant like GPT, to streamline or even automate some of the most tedious tasks and overcome the most important challenges. Among those challenges, one was offering an interesting playground to explore the strengths and limitations of the deployment of AI tools in the industry. But before diving into the description of this challenge, we need to define properly one of Haulogy main application: HAUERP.

1.2.1 HAUERP in details

HAU-ERP is a generic Enterprise Resource Planning (ERP) application developed and sold by Haulogy to its clients. The ERP is exclusively coded in python.

The development framework of HAU-ERP is Odoo, an open-source ERP that exists in two possible versions: a free community version and a paid enterprise version. A new version of Odoo is released every year. In order to keep HAU-ERP stable and relevant for its clients, Haulogy needs to update it to the new version. This task is substantial and poses important challenges.

This thesis aims to explore how machine learning techniques and AI tools can help developers efficiently perform the update process.

The development framework of HAU-ERP is Odoo. Odoo core serves as the core foundation, enabling the creation of modules for a diverse range of functionalities. This framework encompasses a robust API that is consistently provided both with the core and the modules. In the app store, various modules are available, which can be developed by different partners, Odoo or even Haulogy.

Haulogy development process for HAU-ERP leverages the Odoo core, enabling them to craft their own modules while also modifying or merging existing ones to create novel features to cover their own or their clients' needs. For example, Haulogy could personalize the already provided accounting module to better suit the firm or create a module to manage electric meter for one of their client.

Python serves as the primary development language throughout this entire ecosystem. The API, an integral component of this system, eases seamless interaction between the core, HAU-ERP, and the various modules. Odoo framework promote collaboration, adaptability, and continuous development for the HAU-ERP application.

Figure 1.2 shows the conceptual model of HAU-ERP, the different classes of modules and the purpose they serve. Figure 1.3 highlight the dependencies of the different modules. Every arrow denote a dependency. Partner modules, including modules created and shared by Haulogy itself, depend on Odoo core by may also depend on other partner modules. HAU-ERP modules depend on Odoo core or partner modules.

1.2.2 The Migration Process

The development of HAU-ERP is done under a certain version of Odoo, let us call it version v_n. In practice, this version is supported for 3 years after its release. Because Haulogy works with projects that may last longer than those 3 years, updates to new versions have to be performed in order to get advantage of the support.

Whenever a new version of Odoo is released, the updated versions of Odoo and partners modules are provided. Consequently, for Haulogy, shifting from version v_n to v_{n+1} requires a proper migration of

1. The python scripts of any created/modified modules

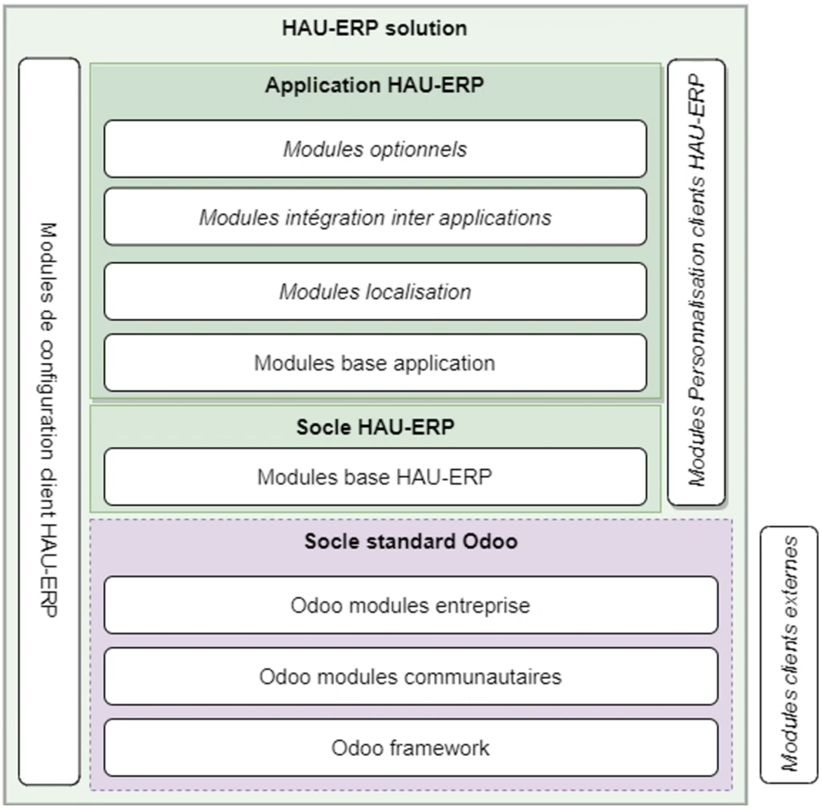


Figure 1.2: Conceptual HAU-ERP diagram.

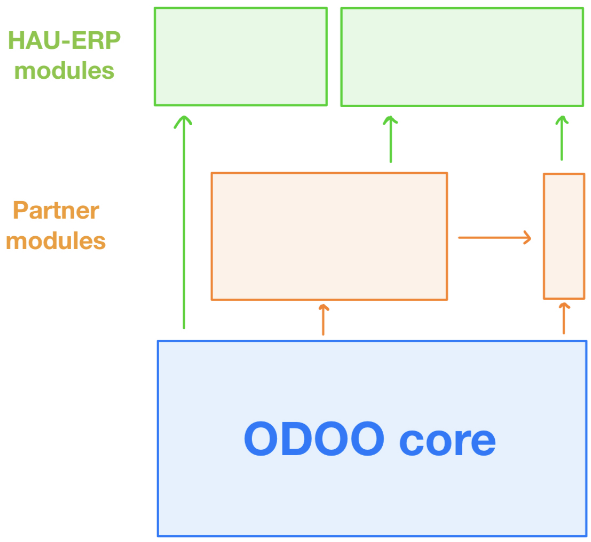


Figure 1.3: HAU-ERP dependencies diagram.

2. The data encapsulated in those modules

The Challenge

Adapting the python scripts of HAU-ERP, the modules in the green part of figure 1.2, for the new version v_{n+1} is an extremely challenging task. As the documentation provided does not list explicitly all the modifications made. Moreover, estimating the impact of a given modification on the whole code is though. Currently, developers have to manually investigate the changes and their impact through trial and error. The python scripts are imported into the new version, bugs are investigated and finally they are corrected. The whole code is frozen during this process, hindering the development of new features.

Data is encapsulated in a SQL database managed by Odoo directly. Thus, the migration of the data to the new version is done by Odoo directly. However, the migration of any added data, such as the data for HAU-ERP modules, have to be handled by Haulogy. This data migration issue, albeit similar to the one exposed just before, will not be the case of interest here.

1.3 Problem Statement

1.3.1 Naive Formulation

To look deeper in the problem and begin exploring solutions, it is important to write a more rigorous statement of the migration process. To achieve this, let us articulate the problem in mathematical terms. Let :

- $\mathcal{X}_n$ be the sequence of codes forming Odoo for version n.
- $\mathcal{Y}_n$ be the sequence of codes forming HAUERP codebase for version n.

The update of HAUERP can be defined as a sequence to sequence translation process

$$f : \mathcal{X}_n \times \mathcal{Y}_n \times \mathcal{X}_{n+1} \to \mathcal{Y}_{n+1} \tag{1.1}$$

Where :

- f is the generative model.

1.3.2 Machine Learning Approach

In the realm of machine learning, the process typically begins with the recognition that there exists a real-world data generation process that produces the data we observe. Here, this process, denoted f, represents the successive modifications applied by the software developers to the sequence of code $\mathcal{Y}_n$. The data consists of the sequences of codes produced, mathematically represented as a sequence $\mathcal{X}$. We seek to approximate this underlying data generation model by training machine learning models on available data. These models aim to capture the inherent patterns, relationships, and structures present in the data, allowing them to make predictions. Through iterative learning and optimization, machine learning algorithms strive to minimize the gap between the modeled distribution and the true data generation process. Therefore, such a model could solve the migration process by automatically inferring the sequence of codes forming the codebase of the new HAUERP version, given the previous codebase of HAUERP and Odoo respectively as well as the one forming the new Odoo version we are updating to.

Formally, we aim at building a model F trained on a dataset $\mathcal{D}$ such that

$$F_{\mathcal{D}} \approx f \tag{1.2}$$

The choice of the dataset $\mathcal{D}$ is crucial. Data is the lifeblood of machine learning, serving as the foundation upon which models are trained. Data provides the context and examples necessary for algorithms to understand the problem space and make relevant decisions. Therefore, the availability and quality of data are crucial, as it directly influence the performance and accuracy of machine learning models.

To train our model, the dataset $\mathcal{D}$ consists of the history of previous HAUERP updates. However, the challenge arises when we examine this history. Often, modifications to the code are not solely driven by the update of Odoo itself, they can be influenced by various other factors. Indeed, a new version of HAUERP is not simply a migration of the python scripts to a new Odoo version, but a whole new project, reflected and engineered for several different objectives. Meaning that several new features may be modified, added and/or removed for the need of Haulogy and/or its clients, but certainly not because of Odoo modifications. Therefore, we have to make a distinction between the migration process and HAUERP update. The former is only a subprocess of the second.

While analyzing the historical data, we might observe changes in the codebase that are seemingly unrelated to the migration process. These changes could be due to bug fixes, feature additions, or even refactoring efforts independently of the update. As a result, using this historical data as a dataset for training a machine learning model introduces unwanted learnt behaviors. Without any prior labeling[1], the model will not be able to differentiate between changes directly tied to the migration and those influenced by other factors, leading to inaccurate predictions and potentially harmful code modifications. Therefore, it is crucial to recognize the complexity and context surrounding code changes and be cautious when leveraging historical data for automated code updates. Without a clear understanding of the underlying reasons for each modification, attempting to train a model in this manner is hazardous and thus not an achievable solution.

Figure 1.4 illustrates the migration process as a Markov Chain. Like before, $\mathcal{X}_n$ is the sequence of codes forming Odoo for version n and $\mathcal{Y}_n$ is the sequence of codes forming HAUERP codebase for version n. To complete this model, variables $\mathcal{Z}_n$ are added. Those are the sequence of modifications to apply to $\mathcal{Y}_n$ and originate from Haulogy internal decisions.

1.3.3 Towards Smart Assistance

As previously mentioned, the main challenge faced by developers during the migration process is the need of a trial and error process to detect the bugs created by the new version. If training a model to perform a fully automatic migration process is not realistic, we can consider a process that generates suboptimal intermediate results to streamline

[1] The current HAUERP transitions history retrievable from the repository commits is too old and large to properly classify the modifications

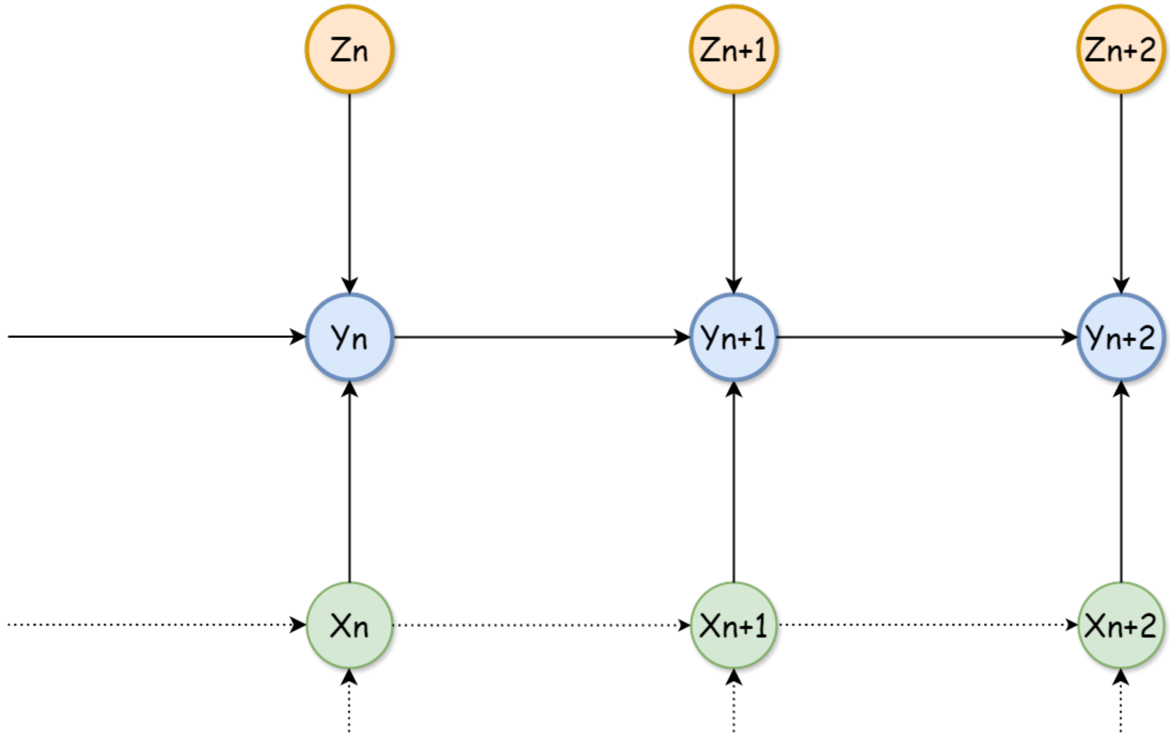


Figure 1.4: HAUERP updates also depends on untracked variables. $\mathcal{Z}_n$ are the untracked decision that influenced the codebase $\mathcal{Y}_n$ of HAUERP.

the complete migration process. In this scenario, the AI agent would serve as an assistant for the software developers.

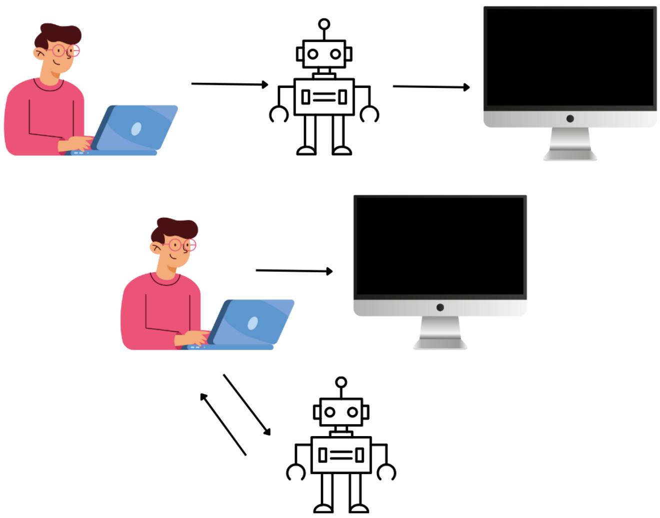


Figure 1.5: The illustration above depicts a scenario in which the AI agent autonomously carries out tasks based on submitted requests. In contrast, the illustration below shows a situation where the AI agent functions as an assistant to the human.

Let us describe an example of how assistance can be significantly helpful.

The lack of documentation provided by Odoo when releasing a new version is one of the main reason developers are struggling to find what parts of their code need to be modified. Therefore, a good assistance lies in the production of an explicit, meaningful and relevant listing of the modifications. For this purpose, we can investigate a model generating a natural language description of the differences between the two versions taken as inputs:

$$f : \mathcal{X}_n \times \mathcal{X}_{n+1} \to \mathcal{T} \tag{1.3}$$

Where $\mathcal{T}$ is a natural language explanation of the difference between the two given codebases.

1.4 Large Language Models

This thesis aims at leveraging Large Language Models (abbreviated to LLMs for the rest of the report) to provide software developers with a proper assistance and relieve them from their most tedious tasks in order to boost their productivity. In this report, we'll dive into the process of building, training, implementing and deploying these models.

A Large Language Model refers to a specific type of Language Model that has been trained on a vast amount of text data, typically on the order of billions or even trillions of words, to understand and/or generate human language sequences. Those models are capable of capturing contextual understanding, and complex language patterns due to their extensive training data and the sophisticated machine learning models used to build them. Recently, LLMs have demonstrated applicable emergent abilities [1] making them a fitting choice to deploy smart assistants that performs automatically generic tasks or ease the more complex ones. LLMs can be used for various natural language processing (NLP) tasks, such as text generation, translation, summarization, sentiment analysis, and more.

Given their extensive capabilities, LLMs excel at being general assistants. Exploiting such a large model to be a useful assistant only when software developers work on the migration process would be a waste of resources. Therefore, in addition to leverage them to provide a relevant assistance in the migration process, our objective extends to providing broader assistance to software developers in their daily coding tasks. This includes tasks such as automatic code synthesis, correction, explanation, etc...

We aim at deploying a general coding assistant. From model selection to deployment, and from training to evaluation, we will meticulously outline each step of our framework.

Chapter 2

State Of The Art

2.1 History of Natural Language Processing

Natural Language Processing (NLP) is a crucial field within artificial intelligence that focuses on enabling machines to understand, interpret, and generate human language. NLP includes a wide range of tasks such as text classification, machine translation, question answering, text summarization, speech recognition, ...

Over time, NLP has evolved significantly, progressing from early rule-based systems [2] to sophisticated deep learning models, which have transformed how machines process language. Recurrent Neural Networks (RNNs) and their variants applied to NLP, such as Long Short-Term Memory (LSTM) [3] and Gated Recurrent Units (GRUs) [4], became the standard for sequence modeling. Those models however struggled to capture long-term dependencies on top of being computationally expensive.

The most significant milestone in modern NLP came in 2017 with the introduction of The Transformer model by Vaswani et al. [5]. By relying solely on attention mechanisms [6], Transformers overcame the limitations of RNNs by enabling parallel processing and better handling of long-range dependencies.

Since then, all state-of-the-art language models have been built upon the transformer architecture.

Before diving into this architecture, we need to understand how a machine processes sequences as inputs.

Tokenization

To be processed by machine learning models, raw text data undergo a fundamental step in NLP tasks: tokenization.

Tokenization is the process of breaking down a text into smaller units called tokens. These tokens can be words, subwords, or characters, depending on the granularity of the tokenization method used.

Tokenization is the first step in text preprocessing pipelines. It prepares the text data for further processing by language models.

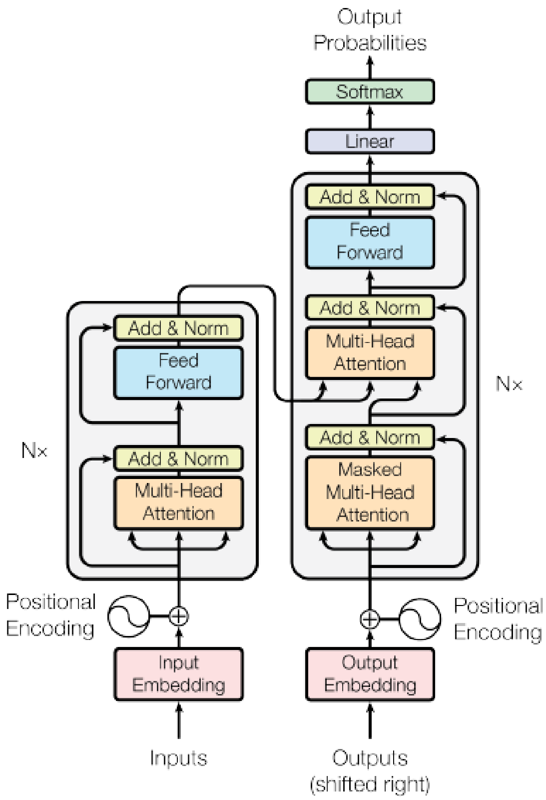


Figure 2.1: The Transformer - model architecture. The left part is the Encoder and the right part is the Decoder. [5]

Modern tokenizers[1] are implemented as objects generating from a sequence of words the corresponding sequence of tokens id each words (or sub-words) corresponds to.

Semantic embedding

Semantic embeddings are numerical representations of words, phrases, or sentences in a high-dimensional vector space where the proximity of vectors reflects semantic similarity between words. Once represented as numerical vectors, the model can apply a various set of operations on the embedded words to produce an output. Historically, these embeddings were generated using models such as word2vec [7] or GloVe [8]. Modern transformer-based architectures, however, use learned embeddings (embeddings learned during training) whose dimensions depend directly on the architecture.

2.2 The Transformer Model

Since its introduction in "Attention is All You Need" by Vaswani et al., in 2017 [5] the transformer model architecture has revolutionized the field of natural language processing by introducing a novel approach to sequence modeling without the need for recurrent or convolutional layers. Instead, the model only uses attention layers (cf figure 2.1).

Understanding the transformer architecture is essential for grasping the underlying principles behind large language models, as many state-of-the-art models, such as GPT series

[1] https://huggingface.co/docs/transformers/main_classes/tokenizer

[9], rely on this architecture as their backbone.

This research will focus exclusively on transformer based architectures. Precisely, decoder-only architecture (more on that later). So, we will thoroughly review each key component of this architecture and its functionalities to understand how modern LLMs process sequences of words.

2.2.1 Model Architecture and Components

Encoder

The encoder stack consists of multiple layers, each containing a multi-head self-attention mechanism followed by position-wise feedforward networks.

Decoder

The decoder stack also comprises multiple layers, but additionally includes multi-head attention over the encoder's output and self-attention over its own input.

Self-Attention Mechanism

At the core of the transformer model are self-attention mechanisms, which enable capturing global dependencies between words in a sequence efficiently. Self-attention allows each word in a sequence to attend to all other words, capturing contextual information effectively. It computes attention scores between each pair of words, determining how much focus should be given to each word when encoding or decoding the sequence. By aggregating information from all words, self-attention enables the model to dynamically understand the relationships between different parts of the input sequence.

The Queries (Q), Keys (K), and Values (V) help the model learn the extent and manner in which words influence each other. This mechanism is formally described in the paper as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V \tag{2.1}$$

Multi-Head Attention

This component allows the model to focus on different aspects of the input sequence simultaneously by projecting the input embeddings into multiple subspaces and computing attention independently in each subspace. This mechanism is formally described in the paper as follows:

$$\begin{aligned} \text{MultiHead}(Q, K, V) &= \text{ Concat } (\text{ head }_1, \ldots, \text{ head }_{\text{h}}) W^O \\ \text{where head }_{\text{i}} &= \text{ Attention } \left(QW_i^Q, KW_i^K, VW_i^V\right) \end{aligned}$$

Where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_{\text{ mod }} \times d_k}, W_i^K \in \mathbb{R}^{d_{\text{mad 1}} \times d_k}, W_i^V \in \mathbb{R}^{d_{\text{moda }} \times d_v}$.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{ head }_1, \ldots, \text{ head }_h) W^O \tag{2.2}$$

Positional Encoding

Since transformers do not inherently understand the order of words in a sequence, positional encodings are added to the input embeddings to provide information about the position of each word.

Feedforward Networks

These are fully connected layers applied independently to each position in the sequence, enabling the model to learn complex relationships between words.

2.2.2 Advantages of Transformers

State-of-the-Art Performance

The transformer architecture has become the model for producing state-of-the-art NLP models, setting new benchmarks across various tasks including language understanding, generation, and translation.

Parallelization

Transformers can efficiently parallelize computations across different words in the sequence, leading to faster training and inference compared to sequential models like recurrent neural networks.

Multi Dependencies

The Multi Head attention mechanism allows the model to learn several different types of word dependencies.

2.2.3 Limitations

Quadratic Complexity

The self-attention mechanism has a quadratic computational complexity with respect to the sequence length, making it challenging to scale to long sequences. Usually, this means that models do not have a large context window. However, while it may have been true for the first released LLMs, latter models displayed the ability to process very large context [10].

Memory Requirements

Transformers require significant memory resources, especially for large models with a high number of parameters, posing challenges for deployment on resource-constrained devices. These memory requirements will pose a significant challenge for the rest of the research. Later we will discuss how they can be reduced effectively.

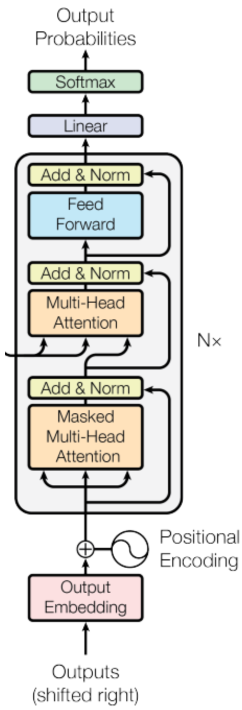


Figure 2.2: Decoder-only architecture. [5]

2.3 From Transformers to Large Language Models

As previously mentioned, the transformer architecture consists of two main components: an encoder and a decoder. In 2018, Radford et al. introduced the Generative Pre-Trained Transformer (GPT) [9], a variant using only the decoder part of the transformer (Fig 2.2). Those types of architectures are referred to as decoder-based or decoder-only.

Decoder-based models focus on the decoder of the transformer. At each stage, the attention layers for a given word can only access preceding words in the sequence. They are well-suited for text generation tasks. The success of decoder-only architectures in generating coherent and contextually appropriate text has been a major factor in developing LLMs for tasks like creative writing, dialogue generation, and code completion.

Encoder-only models also exists, like BERT [11]. Those are more effective for tasks requiring full-sentence comprehension, such as sentence classification, sentiment analysis, and token classification. This research focus exclusively on text generation tasks. Thus, we will focus only on decoder models.

2.4 Mixture Of Experts (MOE)

A new type of architecture has recently gained significance. With the introduction of Mixtral 8x7B and, Mixture of Expert models have become popular in the field of LLMs.

MoE architectures have emerged as a promising approach to scaling deep learning models while optimizing computational efficiency. The core concept behind MoE is dynamic routing of input data to a subset of "expert" neural network layers rather than activating the entire model for each input. This approach allows the model to scale effectively, leverag-

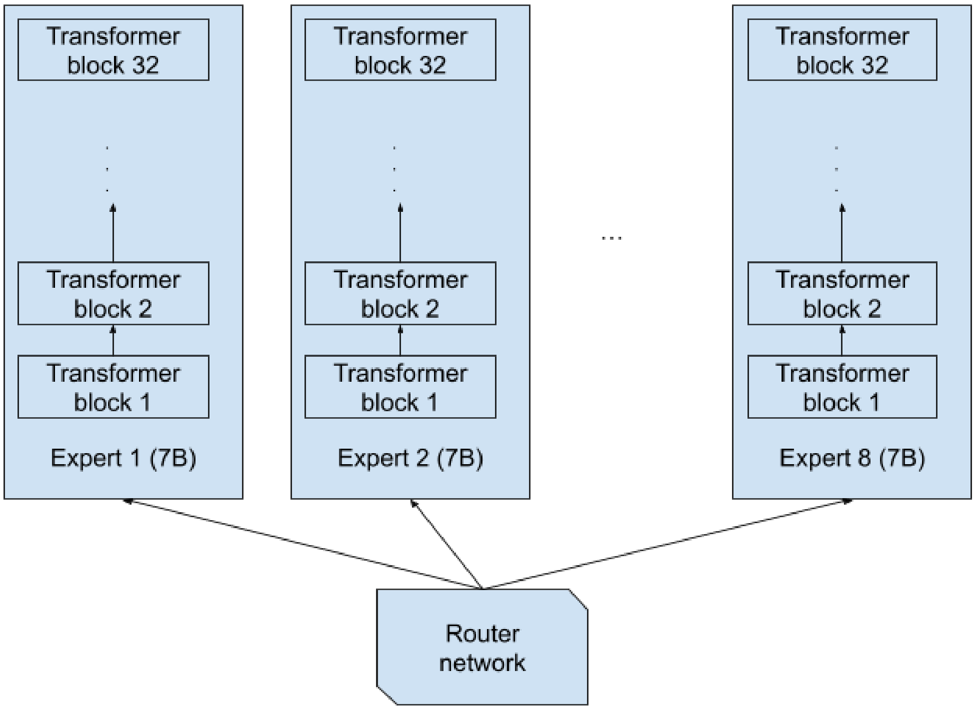


Figure 2.3: Mixtral 8x7B model [12]

ing the benefits of larger model capacity without a proportional increase in computation cost. MoE models have demonstrated state-of-the-art performance in various language task.

Figure 2.3 displays the simplified design of Mixtral 8x7B. At each layer, a router selects two experts to process a token.

2.5 Training Large Language Models

2.5.1 Unsupervised Pre-Training and Supervised Fine-Tuning

But Radford et al. [9] also introduced something important to build large language models; the training process. The model undergo a pre-training phase.

Early transformer models were trained on smaller and more focused datasets. However, for LLMs, models are trained on vast and diverse datasets drawn from the web. This large-scale pre-training allows the models to acquire broad, general knowledge, which can then be fine-tuned for specific tasks or used directly in a wide variety of applications.

LLMs involves a multiphase process that consists of unsupervised pre-training, supervised fine-tuning, and more specifically, instruction tuning. Each stage is crucial in shaping the model's performance and versatility across different tasks. The process is computationally intensive and requires vast amounts of data, often ranging from terabytes to petabytes, as well as large-scale infrastructure, such as powerful GPUs.

Radford et al. [9] mathematically described the task as follows:

Given an unsupervised set of tokens $\mathcal{U} = \{u_1, \ldots, u_n\}$, we try to maximize the following likelihood:

$$L_1(\mathcal{U}) = \sum_i \log P\left(u_i \mid u_{i-k}, \ldots, u_{i-1}; \Theta\right)$$

where k is the size of the context window.

After being pre-trained, models undergo a fine-tuning process where they are trained on supervised dataset to predict according to labels given instruction.

$$L_2(\mathcal{C}) = \sum_{(x,y)} \log P\left(y \mid x^1, \ldots, x^m\right)$$

Where $x^1, \ldots, x^m$ are the set of tokens given as instruction and y the sequence to predict.

2.6 Generation

The output of a decoder-only architecture (or more generally, a transformer) is a probability distribution over all the tokens in the vocabulary for each position. Parameters can be adjusted to sample for this distribution, like the temperature, that squeeze the probability distribution to allow for more hazardous predictions. Sampling the most probable output is known as greedy decoding.

During a forward pass in training, the model predicts multiple tokens simultaneously, but during inference, only the prediction for the final token is relevant, as earlier tokens are part of the input sequence. While generating tokens one by one during inference may seem inefficient, during training, the model predicts all tokens in the context window in parallel, making the process more efficient.

To generate a full sentence during inference, the model first predicts one token based on the input. It then appends this token to the input sequence and uses the updated sequence to predict the next token, repeating this process iteratively.

This approach is why such models are referred to as autoregressive models. The same principle applies whether the model is instruction-tuned or not.

2.6.1 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) [13] represents an approach in LLMs that addresses the limitations of generating responses only based on knowledge acquired during pre-trained. Traditional LLMs, such as GPT and Llama2, rely on their internal parameters to generate responses. However, their fixed knowledge base can become outdated, and scaling them to handle vast amounts of dynamic information can be computationally expensive. RAG offers a hybrid solution by combining the strengths of retrieval-based methods with generative models, allowing for more up-to-date and contextually relevant outputs.

The architecture involves another component on top of the model, the retriever. The retriever is responsible for fetching relevant documents based on the input query. Dense retrievers typically use dual-encoder architectures like those introduced in the DPR (Dense Passage Retrieval) model by Karpukhin et al [14].

The overall process in RAG works as follows:

1. For a given input query, the retriever searches a pre-indexed knowledge base to obtain the top-k relevant documents or passages.
2. The retrieved documents are then passed to the generator along with the original query.
3. The model produces the final output, conditioned on both the query and the retrieved documents.

2.7 Prompt Engineering

Prompt engineering is a critical aspect of interacting with large language models (LLMs), focusing on the creation and refinement of input prompts to optimize the model's output. This process is integral to harnessing the full potential of LLMs, as the structure, wording, and context of a prompt significantly influence the quality, relevance, and accuracy of the generated responses.

Prompts serve as the interface between the user and the LLM, guiding the model toward producing desired outputs. They can range from simple questions to complex instructions, depending on the task. Research has shown that even minor changes in a prompt can lead to significant differences in the output.

Popular prompt engineering techniques involves Zero-shot and Few-shot Prompting.

- Zero-shot prompting involves presenting the model with a task it has not been explicitly trained on, using carefully crafted prompts. Kojima et al. [15] increased the model's precision across various benchmarks by simply adding "Let's think step by step" to the prompt.
- Few-shot prompting, on the other hand, provides the model with a few examples to guide its responses. Brown et al. [16] demonstrated that few-shot learning allows models like GPT-3 to generalize across tasks with minimal examples, significantly improving performance on unseen tasks.

2.8 Evaluation of Large Language Models

2.8.1 Benchmarks

Benchmarks for LLMs are standardized tests designed to assess and compare the performance of these models across various tasks. The primary goal is to quantify the capabilities of LLMs in areas such as natural language understanding, question-answering, text generation, reasoning, **coding**, and more. These benchmarks are essential for guiding model development and assessing progress in the domain.

The most important use of benchmarks is to compare LLMs to one another, as it is yet the only way to state if a model is better than another in a given area.

There exists dozens of benchmarks. In section 3, we only describe the ones that will be useful for this research.

Benchmarks consists of a relatively small set of instruction/reference pairs. The predictions are compared using evaluation metrics characteristics of the benchmark method of evaluation. The value of those metrics is always in a 0 to 1 range. To our knowledge, none of the popular benchmarks has been beaten yet.

2.8.2 pass@k

The pass@k metric, as defined by Chen et al. [17], measures the probability that at least one of the top k generated samples for a given problem is correct. This metric is especially useful for evaluating the performance of models that generate multiple possible outputs for tasks like code generation.

$$\text{pass@ } k := \mathop{\mathbb{E}}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \tag{2.3}$$

The computation require the generation of n candidates to reduce the metric variance.

2.8.3 BLEU

The BLEU (Bilingual Evaluation Understudy) metric is a widely used method for evaluating the quality of machine translation models by comparing the generated translation (hypothesis) against one or more reference translations.

BLEU calculates how many n-grams (continuous sequences of words of length n) in the generated translation match with the reference translations. Commonly, BLEU considers unigrams (1-grams), bigrams (2-grams), trigrams (3-grams), and 4-grams.

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n \right)$$

with

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases}$$

Smoothing has been introduced for this metric to minimize the penalty when calculating the final score for n-grams, particularly when k-grams with k smaller than n are encountered. The classic BLEU metric does not account for these k-grams, but smoothing addresses this by incorporating them into the evaluation.

2.9 Quantization

This section discusses the memory demands of LLMs on GPUs. Each parameter in the model requires memory space. In the common case of using 32-bit floating-point precision (FP32), each parameter takes up 4 bytes. To measure the memory space required to lead a model using FP32, one's simply need to multiply the number of parameters by 4 bytes. For instance, a model with 1 billion parameters would need approximately 4 GB of

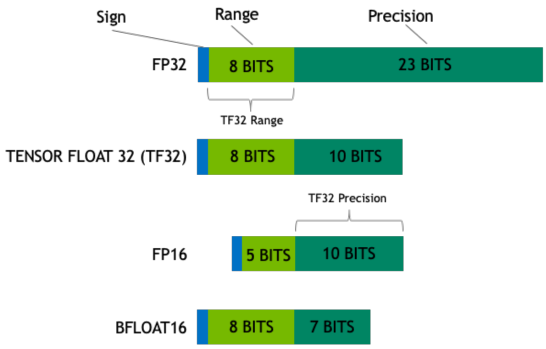


Figure 2.4: Different data types [18]

memory just to store the parameters. Usually, the number of parameters in these models is massive, often ranging from hundreds of millions to tens of billions.

Examples:

- Llama2-70b with 70 billions of parameters requires approximately 28 GB of memory.
- GPT3 with 175 billions of parameters requires approximately 70 GB of memory.

The large size of LLMs arises the need to efficiently manage memory and computational resources. One significant advancement in this area is the shift from FP32 representation to 16-bit representations, which reduces memory requirements by half. Two main 16-bit representation formats exist: FP16 and BF16. The latter sacrifices three bits of the exponent to increase the size of the mantissa, allowing for a greater range of values. (Fig 2.4)

To estimate the memory space (in bytes) required to save a model can be estimated with the following equation:

$$M = (\#P \times 4) \times 1.2(B) \tag{2.4}$$

Where $\#P$ is the number of parameters of the model and the 1.2 factor represents the memory overhead. Q is the amount of bits used to load the model.

Quantization reduces the precision of the model's parameters to 8-bit or 4-bit. Lowering the memory size by two or four. Specifics methods of quantization are described in an upcoming section.

2.10 Parameter Efficient-Tuning

Parameter-Efficient Tuning (PEFT) refers to techniques that fine-tune large language models (LLMs) by adjusting only a small subset of their parameters, instead of updating all the model weights. The goal is to achieve high performance on specific tasks while significantly reducing the computational power requiered compared to traditional full fine-tuning (FFT). PEFT methods are particularly useful for LLMs, where full fine-tuning can be highly expensive due to their size. Especially when considering very large models such as GPT3 [16].

There exist various PEFT methods, and their efficiency is measured by the trade-off between the loss of precision with respect to FFT and the cost saved.

2.10.1 Low-Rank Adaptation

Among those methods, Low-Rank Adaptation (LoRA) [19], a particularly popular method that's widely used for fine-tuning LLMs, offers the best performance and cost trade-off [20].

LoRA insert layers are inserted between existing layers of the model. These additional parameters, called the LoRA adapters, are retrained while the rest of the parameters are frozen. The method assumes that the weight updates during fine-tuning are low-rank, which means they can be expressed as the product of two smaller matrices. It introduces trainable, low-rank matrices into the model's architecture, enabling efficient fine-tuning with minimal additional parameters.

Let W be the parameter matrix. For a pre-trained $W_0 \in \mathbb{R}^{d\times k}$ and its update ΔW

- A forward pass is given by $h = W_0 x + \Delta W x$
- We constrain the update ΔW by representing the latter with a low-rank decomposition $W_0 + \Delta W = W_0 + BA$, where $B \in \mathbb{R}^{d\times r}$, $A \in \mathbb{R}^{r\times k}$, and the rank $r \ll \min(d, k)$
- The forward pass becomes $h = W_0 x + BAx$

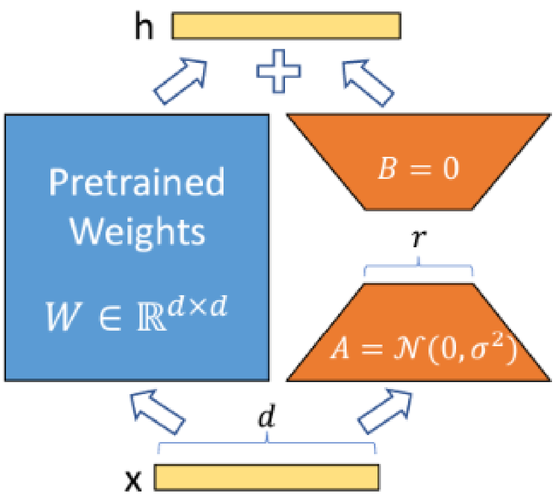


Figure 2.5: Low Rank Adaption: Only the A and B matrix are re-trained. [19]

The low-rank matrices are added at certain layers (often in the attention or feed-forward layers) of existing pre-trained models without requiring any architectural changes, making it easy to implement. The targeted layers can be manually chosen, making the method flexible and adaptable for different case scenarios.

LoRA can achieve comparable performance to full fine-tuning while updating only a fraction of the parameters.

2.10.2 Quantized Low-Rank Adaptation

The QLoRA method introduces three key innovations for training LLMs.

1. 4-bit NormalFloat: A quantization data type that adjusts the block sizes of quantization based on a normal distribution across the entire range.
2. Double quantization: The process of quantization requires a quantization constant, which is itself further quantized.
3. LoRA with quantization: The LoRA technique is applied to a model that has been quantized.

To quantize a 32-bit float tensor into an 8-bit tensor :

$$\mathbf{X}^{\text{Int8}} = \text{round}\left(\frac{127}{\text{absmax}\left(\mathbf{X}^{\text{FP32}}\right)}\mathbf{X}^{\text{FP32}}\right) = \text{round}\left(c^{\text{FP32}} \cdot \mathbf{X}^{\text{FP32}}\right)$$

$$\text{dequant}\left(c^{\text{FP32}}, \mathbf{X}^{\text{Int18}}\right) = \frac{\mathbf{X}^{\text{Int18}}}{c^{\text{FP32}}} = \mathbf{X}^{\text{FP32}}$$

Where c^{FP32} is the quantization constant.

By denoting the A et B matrix of LoRA as L_1 and L_2, the forward pass becomes:

$$\mathbf{Y}^{\text{BF16}} = \mathbf{X}^{\text{BF16}} \text{ doubleDequant } \left(c_1^{\text{FP32}}, c_2^{\text{k-bit}}, \mathbf{W}^{\text{NF4}}\right) + \mathbf{X}^{\text{BF16}}\mathbf{L}_1^{\text{BF16}}\mathbf{L}_2^{\text{BF16}}$$

$$\text{doubleDequant } \left(c_1^{\text{FP32}}, c_2^{\text{k-bit}}, \mathbf{W}^{\text{k-bit}}\right) = \text{ dequant}\left(\text{dequant}\left(c_1^{\text{FP32}}, c_2^{\text{k-bit}}\right), \mathbf{W}^{\text{4bit}}\right) = \mathbf{W}^{\text{BF16}}$$

Upon reducing the number of parameter to train, QLoRA successfully achieves a reduction of memory necessary when training large models with limited memory.

2.11 Large Language Models for Code

LLMs for code, often referred to as coders, are models able to handle coding-related tasks. The primary task associated with coders is code generation, which is typically achieved by pre-training an LLM on a vast corpus of raw code. These models can perform tasks like auto-completion: given the definition of a function, they can complete the function's body. GitHub Copilot is a prominent example of a coding assistant that leverages such models to provide code auto-completion. On the other hand, ChatGPT, while also popular, is instruction-tuned and therefore capable of more than just function generation. It can generate code from natural language descriptions, debug existing code, explain code snippets, and more.

2.11.1 Related Works

Extensive research has been conducted to explore these new capabilities, improve model accuracy, and evaluate performance across various coding tasks.

Chen et al. [17] laid a solid foundation for code generation research by introducing a novel method to assess LLMs' code generation abilities. Their approach emphasizes correctness, using the pass@k metric for evaluation. They also introduced HumanEval, which has become the most widely recognized benchmark for code generation.

Luet al. [21] introduced CodexGLUE, a large supervised dataset designed for training and evaluating LLMs on both code understanding and generation. It features a diverse collection of 10 tasks across 14 datasets.

Muenogiff et al. [22] demonstrated that instruction-tuning their auto-completion model, Starcoder, unlocked new coding abilities. The model, post-tuning, could synthesize code from natural language prompts. Additionally, their study focused on two emergent tasks: code explanation and code fixing. Code explanation involves generating a description of what a piece of code does, while code fixing addresses the ability to debug code. They introduced distinct benchmarks to evaluate these tasks.

In Bhattacharya et al. [23] explored various models' capabilities in generating natural language explanations for given code snippets. They compared evaluation metrics and prompt engineering strategies and fine-tuned models using QLoRA, yielding improved results on test benchmarks.

Yu et al. [24] developed a state-of-the-art family of LLMs aimed at code understanding and generation by fine-tuning on various CodeXGLUE [24] tasks.

Szalontai et al. [25] conducted a comparison of the most popular open-source LLMs for code-related tasks, assessing their performance in code generation and summarization across multiple benchmarks.

Chapter 3

Methods

3.1 Framework

3.1.1 Leveraging The Git Structure

Early in this research, we came up with an approach that aims at exploiting the git structure of the ERP repository. Indeed, Git provides command, like *git diff*, that reports the modification made to the files from one branch (version) to another. The idea is to exploit the ability of LLMs to generate Git commands based on natural language instructions and then translate the command output in natural language explanation.

However, this approach has been quickly abandoned when the tested models generated wrong commands given certain instructions or wrongly interpreted.

We do not elaborate further on the experiments as only a few were run before dropping the idea. However, the insights gained from this failure are important and tailored to the rest of this work.

First, LLMs are not mysterious black boxes, and assuming they possess certain capabilities without evaluation is a wrong assumption. It's essential to establish a robust framework for evaluating a model's accuracy on a given task before claiming it has that capability.

Secondly, focusing on abilities that are actively researched and for which datasets of correct input/output pairs exist is a better starting point. These datasets can be possibly used to fine-tune the models and enhance their performance when necessary.

3.1.2 Leveraging Coding Abilities

To achieve our objective of leveraging Large Language Models (LLMs) for code comparison, our starting point will be Coders, a family of LLMs tailored for coding tasks. However, our aim extends beyond mere coding capabilities. Our work is built upon the foundation laid by Muennighoff et al. [22]. In their study, Muennighoff et al. introduced the concept of extending their coder model, StarCoder, for natural language tasks, through instruction-tuning. Highlighting the potential of instruction-tuning to enhance the capabilities of existing models. In our case, the goal is to deploy a Coder model capable of not only writing code but also correcting it, providing comprehensive explanations for the code it generates or evaluates and most importantly, compare two codes.

We aim to create an assistant that significantly enhances the efficiency and effectiveness of code comparison tasks. This assistant will not only ease the comparison of code snippets but will also provide valuable insights and explanations, thereby empowering developers to make informed decisions and improve the quality of their codebases. Through this innovative approach, we seek to leverage the power of LLMs to advance the field of code comparison and foster greater collaboration and knowledge sharing within the developer community.

This work aims to leverage coders' abilities to provide effective assistance to software developers. The literature describes and evaluates a wide range of these abilities. However, considering the context of this study, we focus on the most relevant coding abilities to exploit.

It is important to note that the terminology used for some abilities can be confusing. For instance, **code explanation** in one publication might be referred to as **code summarization** elsewhere while describing exactly the same task. In, **code explanation** may sometimes refer to translating a code snippet into natural language [23], while in other cases, it involves interpreting a larger piece of code.

Furthermore, we distinguish between generating code by continuing an existing code segment (for auto-completion models) and generating code based on a natural language instruction (for instruction-tuned models). We refer to the latter as **code synthesis**.

While humans tend to differentiate between these tasks, it is possible that LLMs do not make the same distinctions. The relationship between input and output tokens could be learned similarly across tasks, potentially activating the same neural pathways. This hypothesis could be tested later by examining whether the performance of LLMs on these two abilities is correlated.

We believe that establishing a clear and consistent nomenclature is crucial, so we provide a precise definition for each of the abilities discussed. Furthermore, we provide for each of them examples of how they can be used to provide assistance to software developers and boost their productivity. We also specify whether the instructions and outputs are written in code or natural language (nl).

Code Generation (code $\rightarrow$ code)

Code generation involves automatically completing a code given the header of the function (or the class). It can be useful for generating several lines of code quickly, boosting productivity of developers when they work on coding large codes made-up of several functions. GitHub Copilot exploit this ability and has been extremely popular among developers.

Code Synthesis (nl $\rightarrow$ code)

Code Synthesis involves automatically producing code from a higher-level description, specification and/or instruction. For example, given a description like "write a Python function to sort a list of numbers in ascending order," the system can generate the appropriate code. It can be useful for generating entire modules or scripts based on developers requirements.

		Time (GPU hours)	Power Consumption (W)	Carbon Emitted (tCO_2eq)
LLAMA 2	7B	184320	400	31.22
	13B	368640	400	62.44
	34B	1038336	350	153.90
	70B	1720320	400	291.42
Total		3311616		539.00

Figure 3.1: GPU time, consumption and carbon emitted for training the Llama 2 family of models. [26]

Code Explanation (code → nl)

Code explanation is the process describing in natural language what a line of code does. This task is particularly useful for debugging or understanding precisely some codes.

Code Summarization (code → nl)

Code summarization involves condensing a block of code into a brief description or summary that captures the main purpose or functionality of the code. Unlike code explanation, which focus and detail the purpose of one line of code only, code summarization provides an overview.

Code Comparison (code → nl)

Code comparison is the process of analyzing two code snippets to identify differences and translate them in a natural language explanation. To our knowledge, this task has not yet been studied in previous research. In the context of this work, this ability is probably the most useful. Indeed, it is a direct answer to the migration process challenge previously described and could result in an important boost of productivity when developers work on this tedious project.

3.2 Model Selection

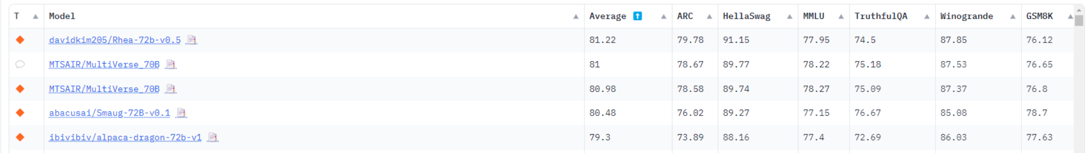
In our approach to deploy LLMs, we use open source models as our starting point. Beginning with open source models for fine-tuning rather than constructing one from scratch is primarily motivated by the significant computational resources required for training large language models.

Indeed, training a state-of-the-art language model from scratch requires a huge amount of data and computational power, with high-end GPUs running for an extensive time. Figure 3.1 display the ressources spent for pre-training the Llama 2 family of models reported by Trouvon et al. [26].

3.2.1 Hugging Face

Hugging Face is a prominent player in the realm of AI generative models, particularly renowned for its contributions to the deployment and use of LLMs. At the heart of the platform offerings is the Transformers library, an open-source library that provides a comprehensive suite of pre-trained models and tools for building, training, and deploying LLMs.

HugginFace also supports the development of LLMs by maintaining a leaderboard of the model achieving the best performances on the most popular benchmarks[1].



T	Model	Average	ARC	HellaSwag	MMLU	TruthfulQA	Winogrande	GSM8K
	davidkim205/Rhea-72b-v0.5	81.22	79.78	91.15	77.95	74.5	87.85	76.12
	MTSAIR/MultiVerse_70B	81	78.67	89.77	78.22	75.18	87.53	76.65
	MTSAIR/MultiVerse_70B	80.98	78.58	89.74	78.27	75.09	87.37	76.8
	abacusai/Smaug-72B-v0.1	80.48	76.02	89.27	77.15	76.67	85.08	78.7
	ibivibiv/alpaca-dragon-72b-v1	79.3	73.89	88.16	77.4	72.69	86.03	77.63

Figure 3.2: HuggingFace leaderboard.

In addition to the Transformers library, Hugging Face provides several tools and services to streamline the deployment of LLMs. This includes the Hugging Face Model Hub, a centralized repository where users can discover, share, and deploy pre-trained models for various NLP tasks. But also datasets easily loaded through their API useful for training or fine-tuning models. Moreover, Hugging Face offers hosted inference APIs, allowing developers to easily deploy LLMs into their applications.

The important availability of models and dataset on this plateform and the strength of its API is the reason we only consider models available there.

3.2.2 Open-Source & Commercial Use

Open-source models allow anyone to inspect and understand how the model was developed, how it operates, and what data it was trained on. This transparency is crucial for building trust in AI systems. Users can verify that the models adhere to ethical guidelines and are safe to use. They also have the freedom to fine-tune and modify the models to suit particular contexts or industries. Open-source licensing ensures that these adaptations can be done without legal or financial constraints.

Using models with open-source licenses that allow unrestricted commercial use ensures that businesses are not caught in legal disputes.

In general, most of the models available in HuggingFace are open-sourced. Nevertheless, some of them comes with conditions of use forbidding any commercial use. We avoid those models.

Also, albeit the model availability, the training dataset is not necessarily available too. However, we accept this scenario as far as the data collection and training process is sufficiently documented.

3.2.3 Limited Computation Power and Financial Resources

The primary objective of this research is to enhance productivity by reducing the number of employees or hours required to complete a task. The benefit to the company lies in

[1]The leaderboard has recently been updated to a new version using different, more modern benchmarks

the potential cost savings. Haulogy will only gain from this research if we successfully implement a model that costs less than the resources it helps the firm conserve.

This imposes a key constraint on our research: rather than pursuing state-of-the-art performances, our primary focus is to achieve satisfying results with minimal cost.

The cost of LLMs arises from two main areas:

1. Training: The computational power required to train a model, which often involves running GPUs for extended periods.
2. Inference: High-end GPUs are needed to load the model and generate responses quickly. Inference must be efficient enough to handle requests from multiple developers simultaneously.

In both cases, reducing the model's memory footprint can lower costs, allowing for the use of smaller GPUs and reducing both training and inference times.

To meet this objective, we **limit our computational power to one single GPU** for both training and inference. Quantization is our primary strategy for reducing memory usage. In an upcoming section, we detail all considerations and techniques involved in making this feasible.

However, it's important to note that quantization can reduce memory usage by at most 4. For example, even with 4-bit quantization, GPT-3 would still require around 17GB of memory just to load. Therefore, we will focus on smaller LLMs with fewer than 10 billion parameters.

This computation constraint is the guiding principle of our research, limiting the range of models we can use and the scope of our experiments.

Furthermore, we are also limited by financial resources. Our experiments will be conducted on Paperspace, a platform offering high-end GPUs at a reasonable cost. However, this platform restricts GPU usage to 6 hours. Those financial constraints are significant, as using a GPU on Paperspace for more than 6 hours incurs additional costs, which we aim to avoid to stay within budget.

3.2.4 Final Selection Criteria

In summary, we select our models among the models available on HuggingFace, unrestricted for commercial use and that comes with a well documented data collection and training process.

Building on the work of [22], we hypothesize that instruction-tuning a coding model unlocks code understanding capabilities (Fig 3.5). If the model's performance in code understanding might come from its instruction-tuning, we hypothesize that it is at least partly due to its original code generation capabilities. Its fluency in writing code likely enhances its comprehension of code-related tokens. Therefore, we will begin with a model that demonstrates state-of-the-art code generation abilities.

Model	Size	Python	C++	Java	PHP	TS	C#	Bash	JS	Avg	MBPP
Multilingual Base Models											
code-cushman-001	12B	33.5%	31.9%	30.6%	28.9%	31.3%	22.1%	11.7%	-	-	-
CodeGeeX2	6B	36.0%	29.2%	25.9%	23.6%	20.8%	29.7%	6.3%	24.8%	24.5%	36.2%
StarCoderBase	16B	31.7%	31.1%	28.5%	25.4%	34.0%	34.8%	8.9%	29.8%	28.0%	42.8%
CodeLlama	7B	31.7%	29.8%	34.2%	23.6%	36.5%	36.7%	12.0%	29.2%	29.2%	38.6%
CodeLlama	13B	36.0%	37.9%	38.0%	34.2%	45.2%	43.0%	16.5%	32.3%	35.4%	48.4%
CodeLlama	34B	48.2%	44.7%	44.9%	41.0%	42.1%	48.7%	15.8%	42.2%	41.0%	55.2%
DeepSeek-Coder-Base	1.3B	34.8%	31.1%	32.3%	24.2%	28.9%	36.7%	10.1%	28.6%	28.3%	46.2%
DeepSeek-Coder-Base	6.7B	49.4%	50.3%	43.0%	38.5%	49.7%	50.0%	28.5%	48.4%	44.7%	60.6%
DeepSeek-Coder-Base	33B	**56.1%**	**58.4%**	**51.9%**	**44.1%**	**52.8%**	**51.3%**	**32.3%**	**55.3%**	**50.3%**	**66.0%**
Instruction-Tuned Models											
GPT-3.5-Turbo	-	76.2%	63.4%	69.2%	60.9%	69.1%	70.8%	42.4%	67.1%	64.9%	70.8%
GPT-4	-	**84.1%**	**76.4%**	**81.6%**	**77.2%**	**77.4%**	**79.1%**	**58.2%**	**78.0%**	**76.5%**	**80.0%**
DeepSeek-Coder-Instruct	1.3B	65.2%	45.3%	51.9%	45.3%	59.7%	55.1%	12.7%	52.2%	48.4%	49.4%
DeepSeek-Coder-Instruct	6.7B	78.6%	63.4%	68.4%	68.9%	67.2%	72.8%	36.7%	72.7%	66.1%	65.4%
DeepSeek-Coder-Instruct	33B	**79.3%**	**68.9%**	**73.4%**	**72.7%**	**67.9%**	**74.1%**	**43.0%**	**73.9%**	**69.2%**	**70.0%**

Figure 3.3: Perfomances of the DeepSeek-Coder family of models on the Multilingual HumanEval and MBPP Benchmarks. Those benchmarks evaluate code generation abilities across different languages including python. [27]

3.3 DeepSeek-Coder

This family of models, introduced by [27], achieves state-of-the-art performances (Fig 3.3) for code generation and reasoning while being very small compared to other LLMs, making it very efficient.

The authors release three models of different sizes (1.3B, 6.7B and 33B), along with their instruction-tuned versions. Due to our memory constraints, only the 1.3B and 6.7B versions of the model are further studied, while the 33B version is excluded.

Model Architecture

DeepSeek-Coder model architecture is a variant of LLaMAForCausalLM. An architecture directly available on HuggingFace API. Like any other LLM designed for auto-regressive generation, it is a decoder-only variant of the transformer model. Figure 3.4 shows a simplified diagram of the architecture. But to better understand this model, let us describe each layer and components in detail, as well as the dimensions chosen for the 6.7B version of DeepSeek-Coder.

1. **Token Embedding Layer**:

   Embedding(32256, 4096): The model uses an embedding layer with a vocabulary size of 32,256 and an embedding dimension of 4,096. This layer converts input tokens into dense vector representations that are passed through the decoder layers.

2. **Decoder Layers (32 layers)**:

   The model consists of 32 stacked decoder layers, where each layer performs the following operations:

   (a) Self-Attention Mechanism:

   - q_proj, k_proj, v_proj, o_proj: These are projection layers for the query, key, value, and output within the self-attention mechanism. All projections are implemented using Linear layers.

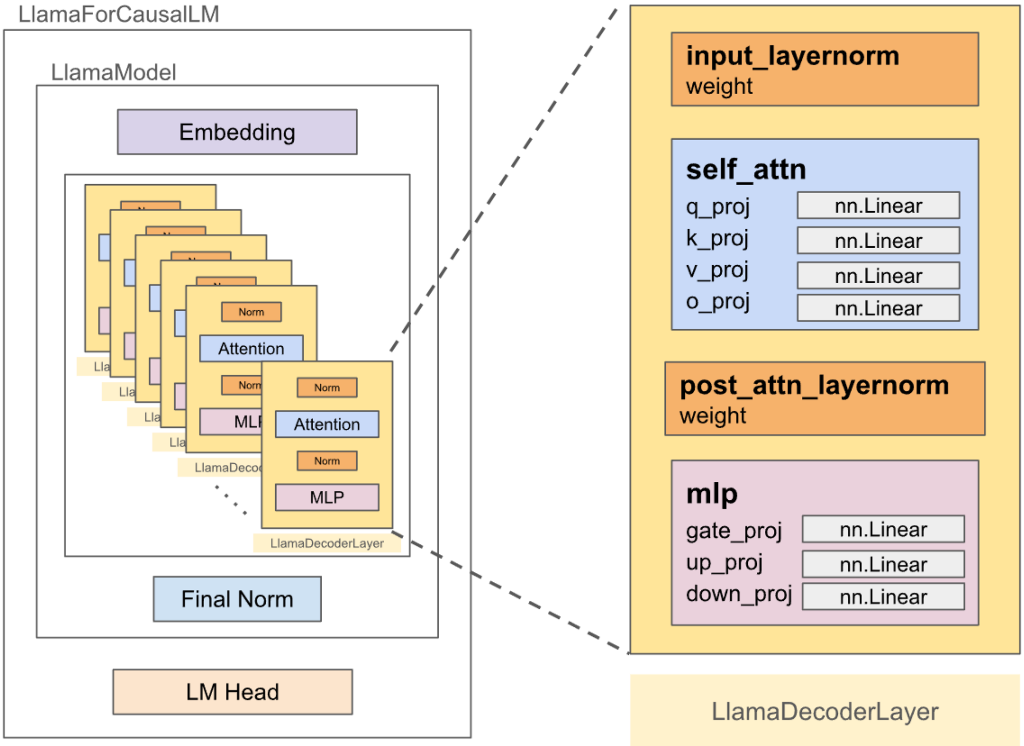


Figure 3.4: LLaMAForCausalLM Diagram [28]. The architecture of DeepSeek-Coder.

- rotary_emb: Rotary Position Embedding. This component handles the rotational positional encoding, which improves the model's ability to capture sequence information.

(b) MLP Block: The MLP (Multi-Layer Perceptron) consists of three linear layers:

- gate_proj: Projects the input from 4,096 dimensions to 11,008 dimensions.
- up_proj: Another projection from 4,096 to 11,008 dimensions, serving as an expansion layer.
- down_proj: Reduces the dimensionality back from 11,008 to 4,096. The activation function used is SiLU (Sigmoid-Weighted Linear Unit).

(c) Normalization Layers:

The model uses Root Mean Square Layer Normalization (RMSNorm) layers for normalization both before and after the attention mechanism and MLP block. RMSNorm is a variant of layer normalization.

3. **Final Normalization Layer**:

The output of the final transformer block is normalized using a global LlamaRMSNorm layer.

4. **Language Modeling Head (LM Head)**:

Linear(in_features=4096, out_features=32256): The final output is passed through a linear layer that projects the 4,096-dimensional hidden states back to the vocabulary size of 32,256. This output is used to predict the next token in the sequence.

Prompt:
You are an AI programming assistant, utilizing the Deepseek Coder model, developed by Deepseek Company, and you only answer questions related to computer science. For politically sensitive questions, security and privacy issues, and other non-computer science questions, you will refuse to answer ### Instruction: *instruction given to the model* ### Response:

Table 3.1: Instruction format for DeepSeekCoder instruction-tuned models.

5. **Key Features**:
    - Rotary Position Embeddings ([29]) : A type of position embedding which encodes the position with a rotation matrix. The angle encodes the positions of the tokens and overall improves the model's handling of large sequence information as their is no limit on the length.
    - Large MLP Expansion (11,008 dimensions): The large expansion ratio in the MLP block (roughly 2.7x) increases the model's capacity to learn complex representations.

To obtain the 1.3B model size, the following modifications are made:

- Reduced the number of Decoder Layers from 32 to 24.
- Decreased the embedding dimension from 4096 to 2048.

Specific Instruction Format

The instructed versions of DeepSeekCoder family of models make use of a specific instruction format (Table 3.1). When given to the tokenizer, a special token is added at the end of the sequence to indicate an "end-of-turn" (or EOT) and simulate a chatting experience. To exploit the most of the model ability to follow instruction, we will stick to this format for any task.

3.4 Towards General Coders Assistant

3.4.1 Framework

The model already achieves state-of-the-art performance, but there are still specific areas we need to explore further. The authors have neither evaluated the model's code understanding capabilities nor assessed the impact of quantization on its performance—two crucial aspects for this research.

Indeed, our goal is to deploy a general-purpose coding assistant that leverages more than just code generation abilities. Building on the work of [22], we hypothesize that instruction-tuning a coding model unlocks code understanding capabilities. In other words, the instruction-tuned models in the DeepSeekCoder family likely possess code

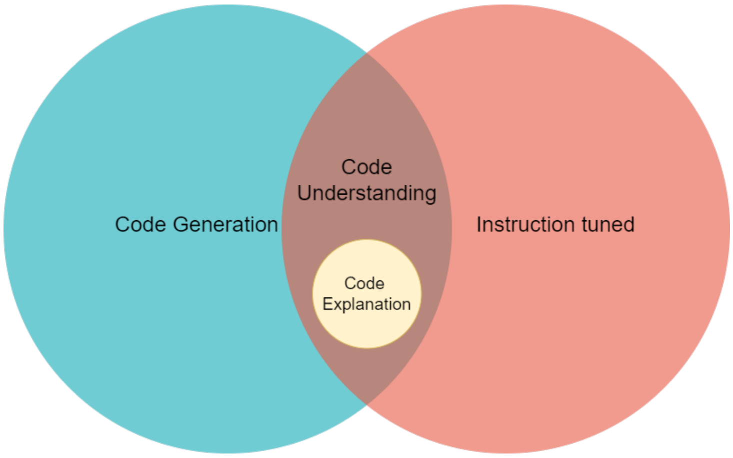


Figure 3.5: When instruction-tuned, coders unlock code understanding abilities.

understanding skills, or at least the ability to summarize code[2] (Fig 3.5).

Given our limited computational resources, we aim to evaluate whether the model maintains strong performance when quantized, ensuring it remains effective if deployed on smaller devices.

Our framework proceeds as follows:

1. We evaluate the code understanding capabilities of **deepseek-coder-1.3b-instruct** and **deepseek-coder-6.7b-instruct**.
2. We quantize the models to assess the performance degradation.
3. We fine-tune the models on different datasets to improve their code understanding abilities.
4. Regarding code generation, the DeepSeekCoder family is already good. So we pursue instead a different objective. We fine-tune the non-instructed models on our local codebase to better align their generations with Haulogy's developer's specific needs.
5. We measure and compare the performance of the fine-tuned models against the original models.

3.4.2 Proof of Concept

We want to demonstrate that the model is capable of performing **code comparison** and illustrate this important task. We build an application to automatically retrieve two modules' python code from Odoo API and give them to the model with the instruction to compare them. The example chosen here is easy. The word advisors have been changed to accountants in the comments and error messages. For such an easy example, **deepseek-coder-6.7b-instruct** is able to spot the differences and make a correct comparison (Table 3.2).

[2] In their paper, this task is referred to as code explanation, but we use our terminology for consistency.

Prompt:
You are an AI programming assistant, utilizing the Deepseek Coder model, developed by Deepseek Company, and you only answer questions related to computer science. For politically sensitive questions, security and privacy issues, and other non-computer science questions, you will refuse to answer ### Instruction: You will be presented with two versions of a Python code and your task will be to describe the difference between the two ### Response: Sure, please provide the two versions of Python code, and I'll be happy to help you identify the differences between them. ### Instruction: First code version : *code at version 15* Second code version : *code at version 16*
Answer:
The difference between the two versions is that the first version is for advisors and the second version is for accountants. The method _autorise_lock_date_changes is also different in both versions.

Table 3.2: Example of a conversation with deepseek-coder-6.7b-instruct to compare two codes. The answer is correct.

3.5 Model Fine-Tuning

On top of evaluating the model's code understanding capabilities, we explore the possibility to enhance them through fine-tuning.

3.5.1 Problem Formulation

The problem of fine-tuning LLMs for NLP tasks can be formulated as follows:

Given:

- A pre-trained LLM with parameters learned from a large corpus of text data through unsupervised learning.
- A specific downstream NLP task.
- A labeled dataset consisting of input-output pairs relevant to the downstream task.

The objective is to:

- Adapt the parameters of the pre-trained LLM to the specifics of the downstream task by fine-tuning the model on the labeled dataset.
- Minimize a suitable loss function.

Loss Function

The loss function to minimize is the cross-entropy loss. This loss aligns the model's predicted token distribution with the ground truth distribution over the vocabulary. The PyTorch documentation [30] describes the implementation of this loss as follows:

$$\ell(x, y) = L = \{l_1, \ldots, l_N\}^\top, \quad l_n = -w_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^{C} \exp(x_{n,c})} \cdot 1\{y_n \neq \text{ ignore\_index}\} \tag{3.1}$$

Where

- C is the number of classes.
- N is the batch size.
- x_n is the input n of the batch.
- y_n is the output n.

Learning Technique

Depending on whether the models are instruction-tuned or not, the training techniques used will differ.

For base models, we employ **next-token prediction**. In this approach, we provide raw code sequences as input and have the model predict the next token for each token in the sequence.

For instruction-tuned models, we use **supervised fine-tuning**. Given an instruction, the model should generate the corresponding output. Although the training process involves feeding the model sequences that include both the instruction and the output, it still performs next-token prediction for each token in the sequence. However, by seeting the labels of the tokens corresponding to the instruction to the *ignore_index*, the contribution of the instruction tokens to the loss is disregarded and the model focus solely on learning from the labels.

Hyperparameters

All training process use AdamW optimizer. The other hyperparameters vary depending on the task involved. The Experiments section details every training configuration.

Padding

When training LLMs, we need to ensure that all sequences have the same length. This is done through padding and truncation. Two opposite operations performed by the tokenizer. The former is and can be set to a fixed length or the length of the largest sequence of the dataset. To perform padding, the tokenizer makes use of a special padding tokens.

Dataset

The dataset chosen depends on the specific task. For each fine-tuning experiment described in Section 4, the exact dataset used is specified.

It is crucial that the dataset accurately reflects the task we aim to improve performance on. Therefore, we use the training set provided with benchmark datasets, if available, to enhance performance for the corresponding benchmark (further details will be provided later).

3.6 Training on a Single GPU

When fine-tuning a large language model on a single GPU, resource limitations like GPU memory and computational power present key challenges. This requires the implementation of strategies to efficiently manage the model's memory footprint and computational load while maintaining training quality. The techniques used and considerations to manage the fine-tuning of large models on a single GPU are a key aspect of our framework.

Parameter Efficient Tuning

To reduce the number of parameters to train as well as the memory size of our models, we make use of QLoRA with a normal-float 4-bit (NF4) quantization and double quantization.

For the retrained parameters, all linear layers are targeted by QLoRA modules, the attention layers and the MLP blocks.

The number of parameters re-trained are the following:

- For the 1.3B version: 19,988,480 parameters are trained.
- For the 6.7B version: 7,495,680 parameters are trained.

Lengths of samples

When training, the samples will be padded to the longest sequence in the dataset. Therefore, it is important to properly pre-process the data and ensure that the longest sequence stays within an acceptable range. By doing so, there is no need to perform truncation.

Batch Size Considerations

One of the first adjustments is reducing the batch size. Due to limited GPU memory, smaller batch sizes are necessary to accommodate the model's parameters and activations during training. Although a smaller batch size can lead to noisier gradients and slower convergence, it is often unavoidable on single GPU setups.

Gradient accumulation can be employed to counter this, effectively simulating a larger batch size by accumulating gradients over multiple steps before performing a backward pass. But this technique only smooth the convergence and does not impact the speed

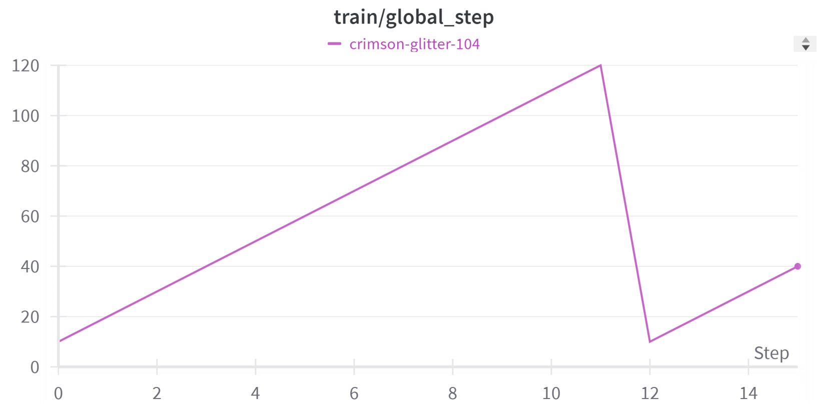


Figure 3.6: A training process restarting with a lower batch size due to memory saturation

of convergence. Therefore, we do not use it to avoid any issues that accumulation could cause.

Because we always train for a limited amount of time, it is better to set the batch to the maximum size possible. But doing this could lead to memory saturation and break the whole training. We use an option available in the trainer of huggingface that is *auto_find_batch_size*. Upon activation, this technique ensures that the GPU never runs out of memory by halving the batch size when the memory is saturated. If not, the process would simply stop.

This approach, however, is not without its flaws. Whenever the memory saturates, the batch size is halved, and the entire process restarts from step 1 (Fig3.6). Although the training state remains unchanged, this implies that if we train for a fixed number of epochs, the overall process becomes longer. Moreover, even if the training could have been completed with a slightly smaller batch size, this method always halves it without ever-increasing it again (cf Appendix B).

Therefore, it is crucial to select the right batch size to accelerate the process while minimizing the risk of memory saturation during training.

Gradient Checkpointings

Gradient checkpointing (Chen et al. [31]) is another crucial technique to manage memory. It reduces memory usage by trading off some recomputation for lower storage requirements. Instead of storing the activations of all layers (high-dimensional outputs produced at each layer) during the forward pass, only certain checkpoints are saved. During backpropagation, intermediate activations are recomputed as needed, leading to a significant reduction in memory usage. While this increases the computation time slightly, it allows the model to be trained with larger batch sizes or on larger models than would otherwise be feasible.

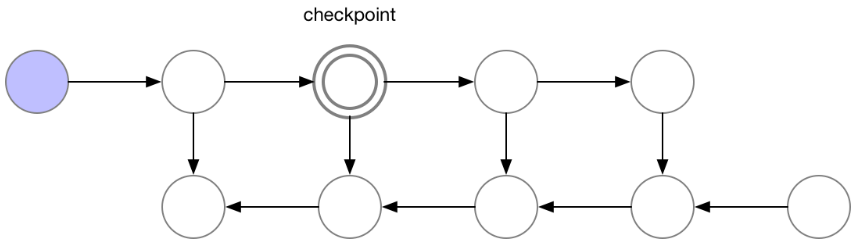


Figure 3.7: Gradient checkpointing. [32]

Minimum Memory Requirements

Taking into consideration everything mentioned above, the 6.7B and 1.3B models can be fine-tuned on a single GPU with as little as 16GB of VRAM[3].

Best Configuration

Appendix B presents a failed attempt at fine-tuning. After multiple trials, we concluded that given the significant constraints on computational power, it is more effective to train on a very small dataset. Although this may lead to overfitting (as discussed in Section 4), it at least allows the model to undergo training and reduces evaluation loss, even if only slightly. This enables evaluation and further discussion.

3.6.1 Training and Testing set

Evaluation of our LLMs performances across all the chosen coding tasks is an essential aspect of our methodology. To properly assess the performances of our model, we make use of benchmarks. For each task, one benchmark is chosen.

Benchmark often comes from existing dataset that we can split in a training and a testing set. Using the testing set as a benchmark and the training set for fine-tuning.

HumanEvalSynthesize

This benchmark is an extension of HumanEval [17], a benchmark that evaluate code generation ability for python code using pass@k metric, by Muennighoff et al. [22] to other coding languages. Python being the only language of interest in the context of this work, we might conclude that this benchmark is no different than HumanEval. But this benchmark is still useful, because its implementation also corresponds to a version of HumanEval adapted for instruction-tuned models.

HumanEvalExplain

Introduced by Muennighoff et al. [22], this benchmark aims at evaluating the code summarization abilities of the model by using the same approach as HumanEval and leveraging the pass@k metric.

The idea is simple:

[3]We haven't tested with less VRAM

1. The model generates an explanation given a code (the code samples are the same as HumanEval).
2. The model generates a code given as instruction the explanation he generated right before.
3. The generated functions correctness

The strengths of this benchmark are similar to those of HumanEval, as it focuses on the correctness of the generation rather than comparing it to a ground truth. This approach is valuable because there are multiple ways to explain a piece of code.

However, a limitation of this benchmark is that the performance is constrained by the model's code generation capabilities.

$$Code \underset{CS}{\longrightarrow} NL \underset{CG}{\longrightarrow} Code \tag{3.2}$$

When evaluated from a sample, the model first performs code summarization (CS) and then code generation (CG). Consequently, we cannot rule out the possibility that an erroneous answer could be due to a mistake made during the code generation phase.

We accept this limitation of the benchmark because our main goal is to compare different models and identify which one is better. We are not focused on developing state-of-the-art models, so we are willing to accept if the performance evaluations are underestimated.

For all HumanEvaluation, we perform greedy sampling and thus measure the pass@1 performance.

While experimenting, we observed that the HumanEval family of benchmarks performance is dependent on the execution environment and might be underestimated. To ensure a fair comparison, we evaluated all models in the same environment.

CodeXGLUE

The CodeXGLUE dataset is a collection of evaluation datasets and tasks designed to assess the performance of machine learning models on code-related tasks. It covers a variety of programming languages and challenges, including code summarization, code completion, code translation, and code clone detection, among others. It comes with a testing set to use as benchmarks for evaluation.

For our purposes, we focus exclusively on the code summarization task. This means that CodeXGLUE evaluates the same task as HumanEvalExplain, but uses a different approach. Specifically, it uses a variant of the BLEU score named CodeBLEU [33] to measure the similarity between a generated code summary and a ground truth. This variant takes into account the syntax similarity to further accommodate to code.

One advantage of this approach is its simplicity, which allows for the creation of a large dataset—over 10,000 samples for benchmarking alone, with hundreds of thousands more available for training. However, this method can lead to a situation where a summary might be rated poorly if it deviates significantly from the ground truth, even if it is perfectly correct.

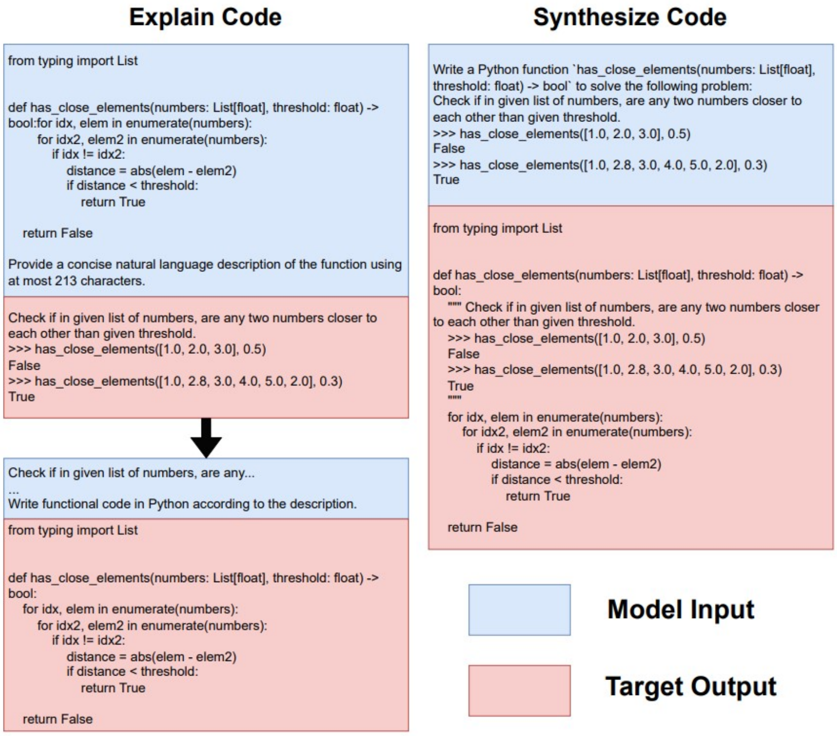


Figure 3.8: HumanEvalExplain and HumanEvalSynthesis illustrated. [22]

We conduct experiments to compare this benchmark with HumanEvalExplain to determine if those two evaluation methods and metrics agree on the performances.

The first 200 samples are selected from the testing set. Due to computation power limitation, it would take too long to evaluate on the whole set.

Conala

The CoNaLa (Code/Natural Language Challenge) dataset [34] is a collection of natural language (NL) to code examples, specifically designed to aid in the development of systems that can translate natural language descriptions into Python code. It contains over 2,900 manually curated NL-code pairs, with an additional 600,000+ examples mined from open-source repositories. The dataset is often used for training and evaluating models in the domain of code generation, and is particularly valuable for tasks that involve converting user intents expressed in natural language into executable code. An example of code generation.

Conala involves 2-shot prompting to generate answers. Two examples of correct instruction-output pairs are provided along with the instruction to enhance performances.

Prompt:
Instruction:examples['instruction1']
Solution:examples['solution1']
Instruction:examples['instruction2']
Solution:examples['solution2']
Instruction:text
Solution:"
Answer the following instructions in one line of Python code:

We use the training set to fine-tune the model for code explanation by swapping the original inputs with the outputs and adapting the instructions accordingly.

This method might initially seem confusing because:

- Evaluating on the Conala testing set (benchmark) assesses the model's code generation capabilities.
- Fine-tuning on Conala training set aims at enhancing code explanation abilities.

This approach might seem confusing at first, but since the two tasks are opposites, it can provide valuable insights into how different coding abilities are correlated.

CommitPackFT & CompareEval

This dataset, which contains approximately 50,000 commit samples, was originally created to instruction-tune the Starcoder model [22]. However, the dataset's features are also suitable for building a code comparison dataset as we have defined it. Each sample includes a piece of code, an updated version of the code, and a commit message describing the changes. Exactly the type of task we're interested in.

After pre-processing the dataset (as detailed in the experiment section), we select the first 100 samples to create the benchmark. We name this benchmark **CompareEval**.

The remaining samples will be used for fine-tuning for code comparison. There's no specific reason for selecting the first 100. Initially, the selection strategy was random, but choosing the first 100 offers the benefit of easily reviewing this portion of the data on HuggingFace data viewer.

To compute a score for the benchmark, given the presence of a ground truth, it is straightforward to compare the generated answer using the BLEU metric with 4-grams, similar to the approach used in Conala. Compared to the pass@k metric, this approach relies on a ground truth, shifting the focus from evaluating the correctness of the comparison to simply matching the expected output. While there exists in fact several ways to compare codes. However, it is easier to implement, as using the pass@k metric would require manually preparing a dataset with test units for each sample.

Nevertheless, it's worth noting that a different approach, similar to HumanEvalExplain, could have been considered. In this scenario, the model would generate a natural language explanation of the differences between the two code versions. The model would then be required to rewrite the new code based on the old code and the generated comparison.

The generated function could be validated using the tests provided with the samples, ensuring that the function is correct.

Prompt:
You will be given two Python code snippets: one is the original version, and the other is the updated version. Your task is to provide a clear, concise, and accurate short description of the update that was made in the updated version. Now, here are the original and updated code snippets for you to analyze: Original code: old_code Updated code: new_code

Chapter 4

Experiments And Results

4.1 Fine Tuning

This section details all the fine-tuning processes runned on different models of the DeepSeek-Coder family.

Every process pursue a different objective, detailed at the beginning of each corresponding sections.

The models of GPU used vary according to the computation power required for the training.

4.1.1 Fine Tuning on Local Code

Objective

This experiment treat of the fine-tuning of **deepseek-coder-1.3b-base** and **deepseek-coder-6.7b-base**, the non-instructed version of the models, on a local code base.

The goal is not to enhance the code generation capabilities of the models, but rather to train the model on private codebases to increase its contextual understanding and better align it with your organization's specific needs. This allows the model to become familiar with your organization's internal conventions, libraries, classes, and more.

Data Collection & Preprocessing

The primary source of Data is Odoo GitHub repository, which contains all python codes of the Odoo ERP across various versions. We only select one version, the version 15.0 which is the one currently used by HAUERP. On top of it, the HAUERP repository also serves as a strong basis for the dataset. By scraping these two large repositories, we retrieve and store the content of all python scripts.

Upon examination of the lengths of the sequences in the dataset, before and after tokenization (Figure 4.1), we notice that some samples are excessively long. These samples far exceed DeepSeek-Coder's context window. In fact, HAUERP contains several scripts that define large dictionaries of data, leading to these oversized samples. These codes are not relevant since we do not intend to train the model to predict specific data values.

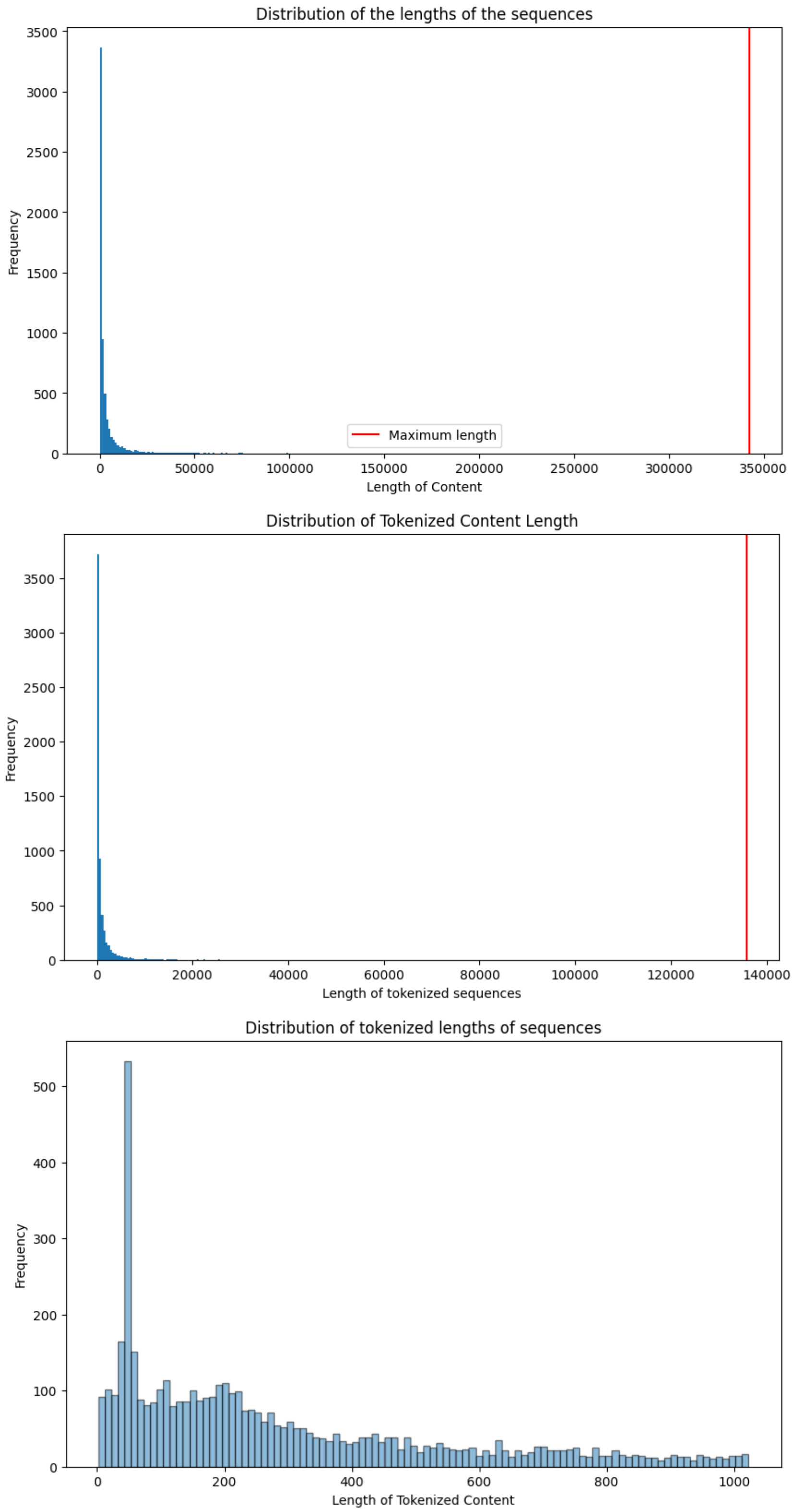


Figure 4.1: The distribution of the lengths of the codes in the dataset. Some samples are too large, even after tokenization. After filtering, the maximum tokenized length in the dataset is 1024.

Prompt:
{raw code sample}

Table 4.1: Prompt used for fine tuning on local code. No instructions are provided.

During training, all sequences are padded to match the length of the largest sequence in the dataset. If these large sequences are not filtered out, it would result in every sequence being padded to the maximum size, way too large here. Although truncation is an option, it would lead to incomplete sequences of code being used for training, potentially undermining the training process.

To ensure a smooth and efficient training process, the length of the sequences should remain within a reasonable limit. Therefore, we set the maximum acceptable length of the sequence to 1024 tokens and filtering the dataset accordingly. This approach allows us to retain a significant portion of the dataset (4790/6180 $\approx$ 77%) while eliminating problematic samples.

Training configuration

In this case, the dataset lacks any guidance or predefined input/output pairs. Therefore, we fine-tune the auto-completion models (the non-instructed ones), and the models are trained using a next-token prediction approach. The prompt consists of feeding the raw code content directly into the model 4.1.

Hyperparameters :

- Optimizer: AdamW ($\beta_1 = 0.9$, $\beta_2 = 0.95$)
- Learing rate: $5e - 4$
- Warming up steps: 50
- Batch size: 8
- Number of epochs:
  - 4 for **deepseek-coder-1.3b-base**
  - 7 for **deepseek-coder-6.7b-base**

Results

Figure 4.2 displays the training and evaluation loss curves for the fine-tuning of **deepseek-coder-1.3b-base**.

Figure 4.3 displays the training and evaluation loss curves for the fine-tuning of **deepseek-coder-6.7b-base**.

After only one epoch, *deepseek-coder-6.7b-base* and *deepseek-coder-1.3b-base* already overfit the evaluation set. Considering the relatively small size of the dataset, this result is not so surprising.

This is probably due to the models being already really good at code generation. The training loss curves plummet quickly, the models learn fast. Thus, it does not take long

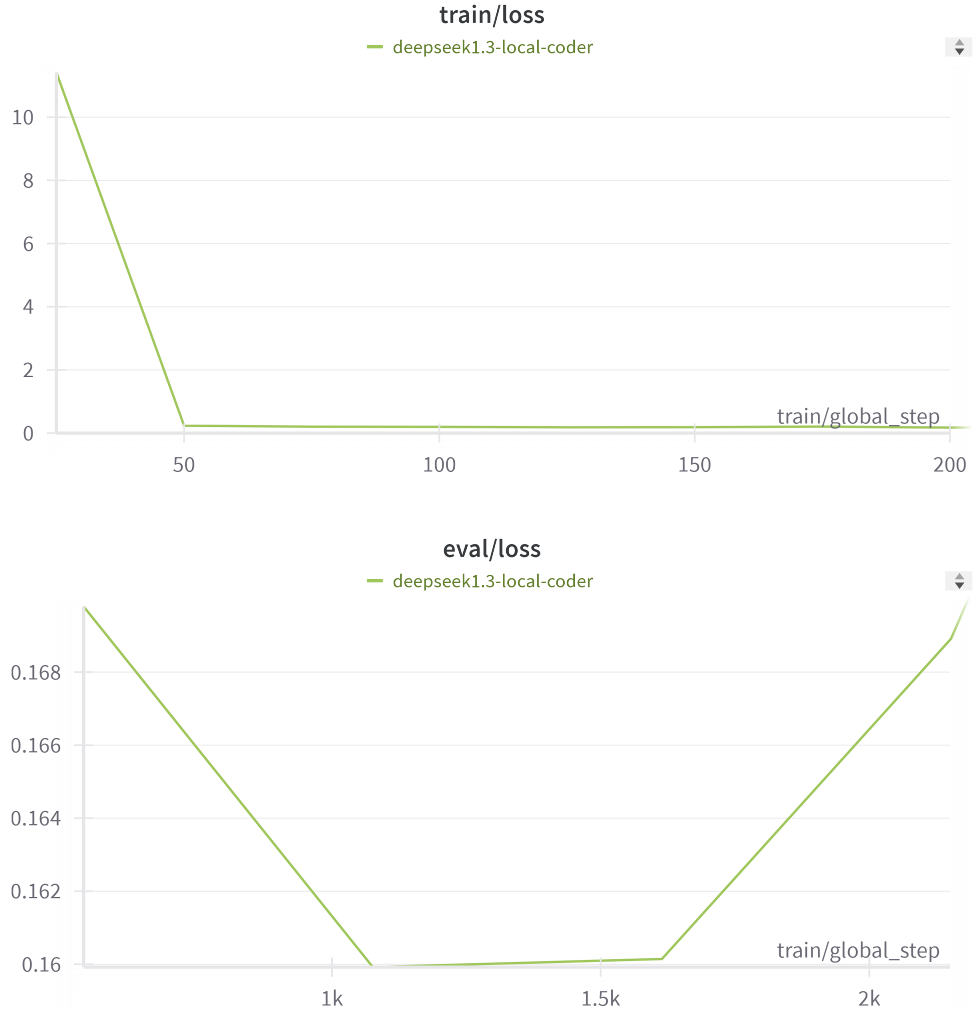


Figure 4.2: Loss curves obtained when fine-tuning deepseek-coder-1.3b-base on a local code. Above: the loss over the training set. Below: the loss over the validation set.

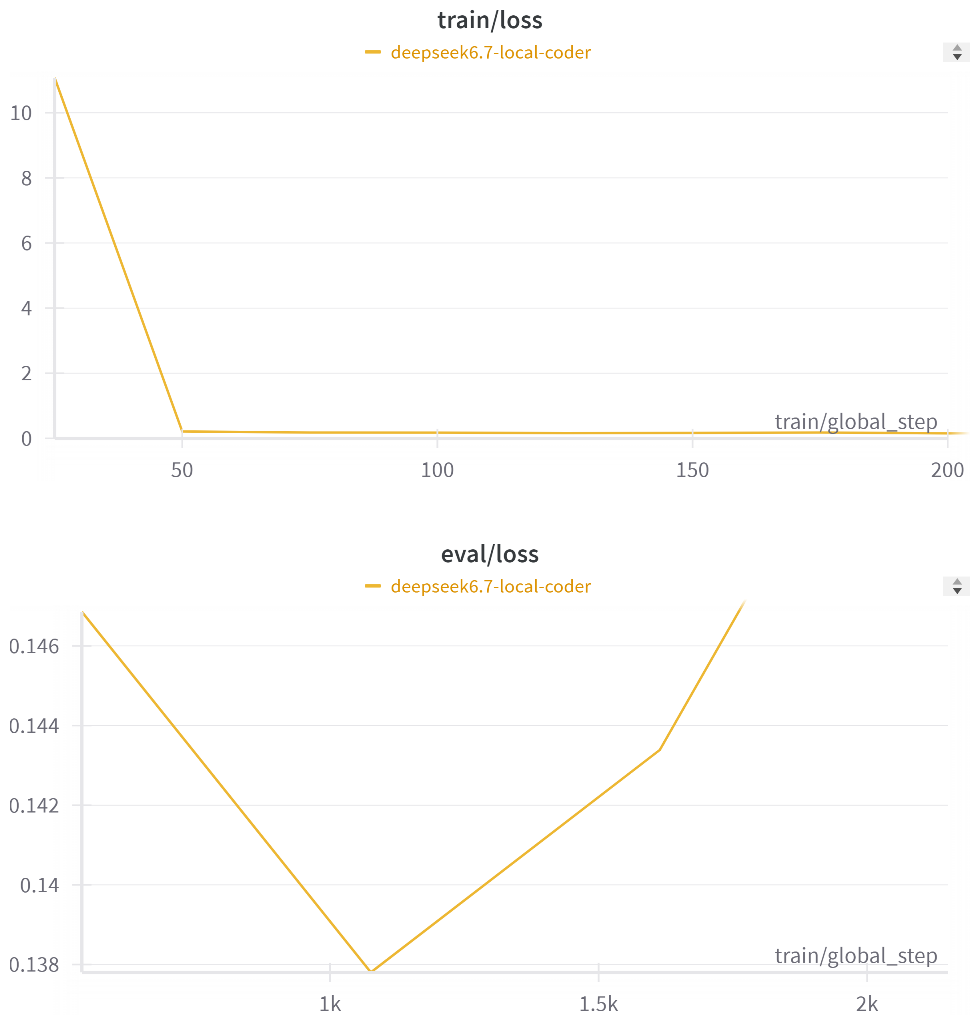


Figure 4.3: Loss curves obtained when fine-tuning deepseek-coder-6.7b-base on a local code. Above: the loss over the training set. Below: the loss over the validation set.

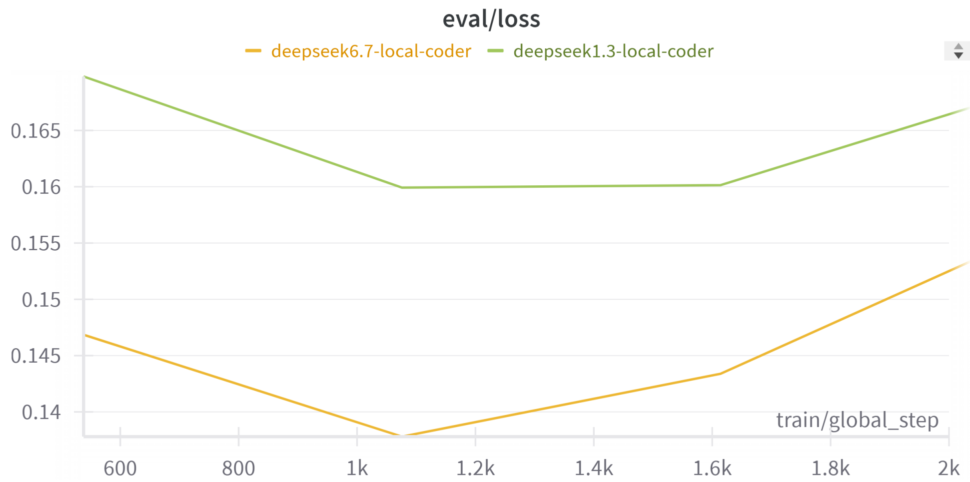


Figure 4.4: Loss curves over the validation set when fine-tuning the two models. Deepseek-coder-6.7b-base achieves the minimal loss.

	Training split	Testing split
Number of samples	2380	500

Table 4.2: Distribution of the number of samples across the dataset.

for the models to overfit.

Also, we cannot reject the possibility for the DeepSeek-Coder family of models to have already been pre-trained on Odoo repository.

Out of the two models, **deepseek-coder-6.7b-base** achieves the minimum loss over the validation set (Fig 4.4). For this reason, the next fine-tuning experiments are conducted only on this model.

We save both models after one epoch of training and name them respectively **deepseek6.7-localcoder** and **deepseek1.3-localcoder**. They are pushed to the Hub but kept private as they are trained on confidential data. To access them, a special permission has to be granted.

4.1.2 Fine Tuning for Code Explanation

Objective

The objective is to train **deepseek-coder-6.7b-instruct** on the training split of the conala dataset to enhance its precision for the code explanation task.

Data Collection & Preprocessing

The dataset is directly loaded from HuggingFace Hub. No pre-processing is done because the number of samples is already pretty small (Table 4.2) and made-up of small sequences.

The training split is used as the training set. The dataset is too small to further split the training split into a training and an evaluation set so the testing split is used as the evaluation set. The testing split is also used for the benchmark. While it might not be ideal to have the same evaluation and testing set, both use different metrics to evaluate the model. This is thus an opportunity to explore how the cross-entropy loss relate to the bleu score computed on the benchmark.

Training configuration

The prompt is made up of the instruction prompt format given for the instructed DeepSeek-Coder models. The {code_snippet} corresponds to the *snippet* feature and the {output} to the *rewritten_intent* feature (See 4.3).

Prompt:

```
You are an AI programming assistant, utilizing the DeepSeek Coder model,
developed by DeepSeek Company, and you only answer questions related to computer science.
For politically sensitive questions, security and privacy issues,
and other non-computer science questions, you will refuse to answer.
### Instruction:
Below is a line of python code that describes a task.
Write one line of summary that appropriately describes the task that the code is performing.
{code_snippet}
### Response:
{output}
<|EOT|>.
```

Table 4.3: Prompt used for fine tuning on Conala dataset.

The model is trained in a supervised-learning fashion, and the model is evaluated on its prediction for the {output} only.

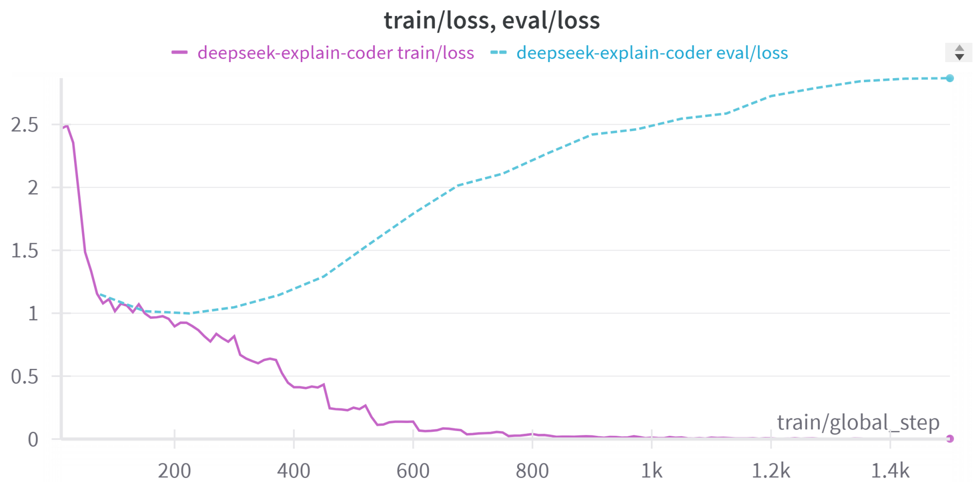
Hyperparameters:

- Optimizer: AdamW ($\beta_1 = 0.9$, $\beta_2 = 0.95$)
- Learing rate: $1e - 4$
- Warming up steps: 50
- Batch size: 32
- Number of epochs: 20

Other configurations were tested, but the optimizer remained unchanged. We report only the configuration that yielded the best results.

Results

Even though we manage to converge properly for the training loss, the model actually overfit after four epochs (Fig 4.5). We hypothesize that this is again due to the small size of the dataset. However, conversely to the overfitting occuring when fine-tuning on local code, the minimum achieved for the evaluation loss is large.



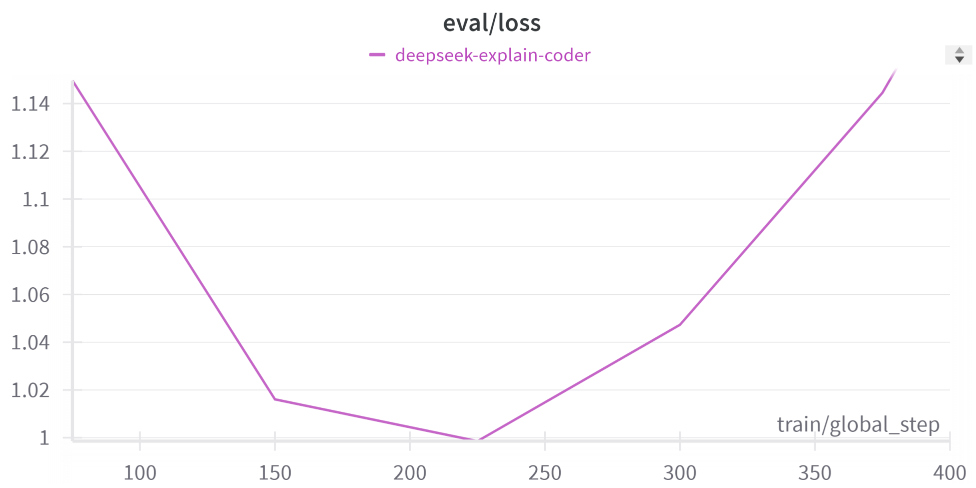


Figure 4.5: Loss curves obtained when fine-tuning deepseek-coder-6.7b-instruct on Conala. Above: the loss over the training set and validation set. Below: the loss over the validation set zoomed in.

Nonetheless, we assume that this is the best we can achieve with our configuration and save the model for evaluation to observe if any improvement was made.

The training is early stopped, the model is saved after 3 epochs and pushed to the Hub. We name this fine-tuned version **deepseek6.7-explain-coder**.

4.1.3 Fine Tuning for Code Comparison

Objective

The goal is to develop the most effective model for code comparison, ensuring it achieves a level of precision sufficient for deployment in assisting the migration process.

Data Collection & Preprocessing

This task requires the model to process not one but two codes as input. If one or the two of them are long, then we might exceed DeepSeek-Coder context window size. And even if we do not, we want to keep a reasonable length of sequence as input for training.

To anticipate the size during training, we build samples by concatenating the old and new code. We also add the prompt instruction format but provide no instruction as it often consist of one sentence only so this does not impact so much the final size.

Figure 4.6 shows that the samples are already pretty small with no real outliers as what had been observed for local codes. Nonetheless, most of the samples being below 1024 length after tokenization we still filter so we can gain memory during fine-tuning. Out of 56025 samples in total, we end up with 55827, preserving more than 99% of the dataset.

Finally, 5000 samples are randomly selected from the filtered dataset to train on. This subset is further split into a training and validation set.

Results

The training loss converges to 0, but the model actually overfit after only one epoch (Fig 4.7). And the minimum achieved for the evaluation loss is large.

Nonetheless, we assume that this is the best we can achieve with our configuration and save the model for evaluation to observe if any improvement was made.

The model is saved after 1 epoch and pushed to the Hub. We name this fine-tuned version **deepseek6.7-compare-coder**.

4.2 Models Performances

This section is dedicated to evaluating the performance of the models on the selected benchmarks.

We begin by examining the performance of the DeepSeek-Coder family of models before fine-tuning.

The authors of DeepSeek-Coder did not assess the code understanding capabilities of their models. However, these results are crucial for our framework, as they establish a baseline

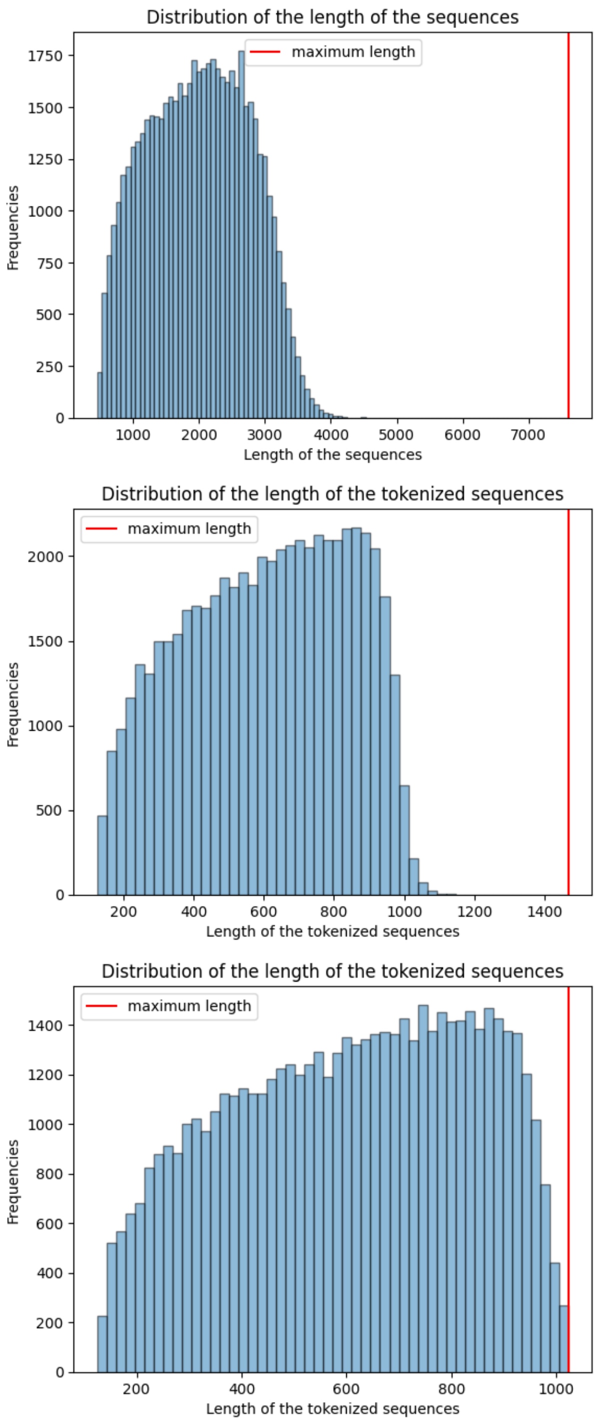
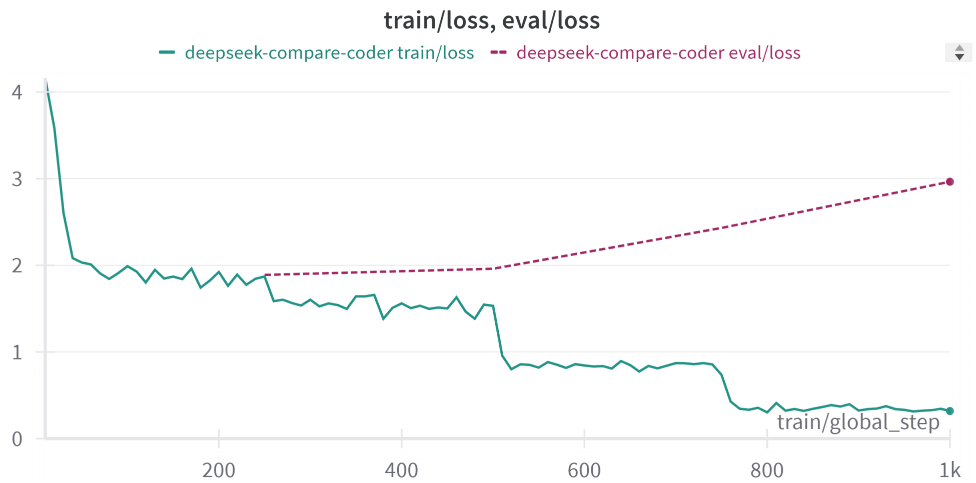


Figure 4.6: The distribution of the lengths of the codes in the dataset. The samples already have a small length, but filtering is still applied to gain memory during training.



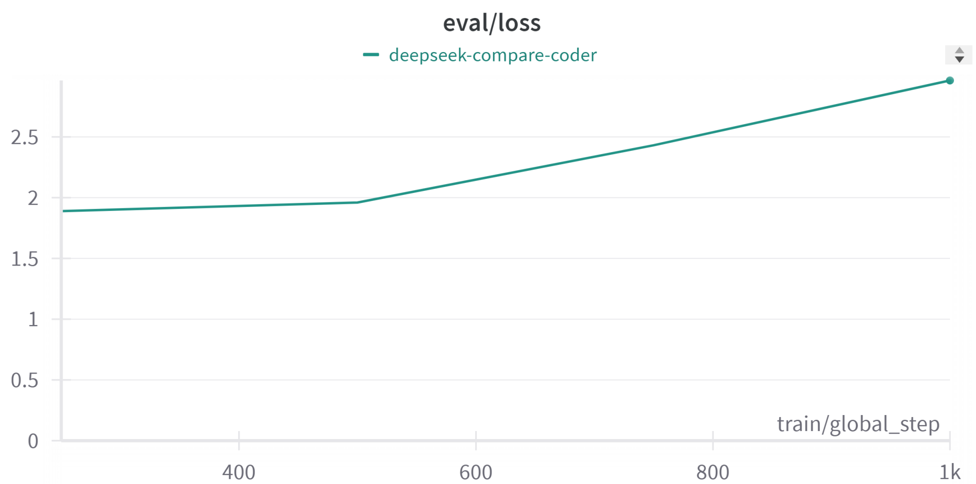


Figure 4.7: Loss curves obtained when fine-tuning deepseek-coder-6.7b-instruct on Conala. Above: the loss over the training set and validation set. Below: the loss over the validation set zoomed in.

for performance. This allows us to determine whether the fine-tuned versions offer any improvement.

Additionally, we investigate the impact of quantization on the performance of these models.

The fine-tuning processes produce a total of 4 new models :

- **deepseek1.3-localcoder**
- **deepseek6.7-localcoder**
- **deepseek6.7-explain-coder**
- **deepseek6.7-compare-coder**

Our goal is to compare the performance of these models across all benchmarks, both against each other and against their non-fine-tuned counterparts.

4.2.1 Effect of quantization

To measure the impact of quantization on the models performances we evaluate **deepseek-coder-6.7b-instruct** and **deepseek-coder-1.3b-instruct** with their 8bit and 4bit quantized versions on both HumanEvalSynthesis and HumanEvalExplain.

Results

Figure 4.8 and 4.9 displays respectively the performances measured on HumanEvalExplain and HumanEvalSynthesis for **deepseek-coder-1.3b-instruct**, **deepseek-coder-6.7b-instruct** and their quantized versions.

In both scenarios, quantization preserves most of the models' performance. Interestingly, 8-bit quantization of **deepseek-coder-1.3b-instruct** even outperforms the original model.

For code synthesis, both models can be safely quantized to 8-bit without any degradation in precision. This implies that the memory requirements for deploying the DeepSeekCoder family are low.

Nonetheless, the execution environment dependency is evident here, as the 1.3b model does not exhibit the same performance as the self-reported results 3.3.

The exact pass@1 measurements are reported in 4.4.

4.2.2 CodeXGLUE: Code Summarization with BLEU

CodeXGLUE benchmark evaluates a model on the same task as HumanEvalExplain, code summarization. The performances of **deepseek-coder-6.7b-instruct** or **deepseek-coder-1.3b-instruct** on this benchmark are reported in table 4.5. The 1.3B model achieves better performance, indicating that the smaller model outperforms the 6.7B model. This result contradicts our previous results as well as the HumanEvalExplain results, which emphasizes correctness and is therefore considered more reliable.

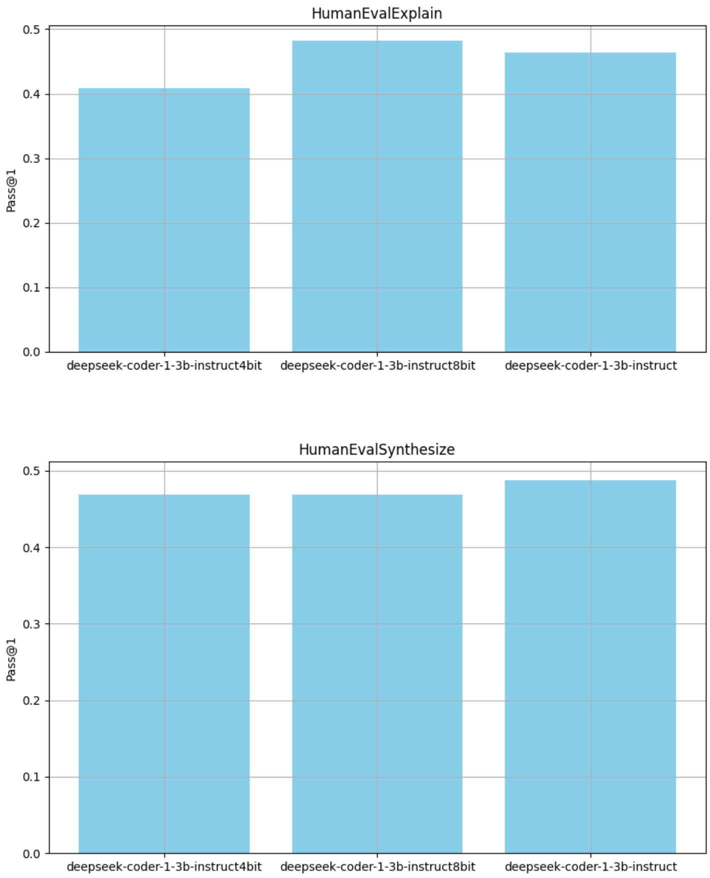


Figure 4.8: Effect of quantization on deepseek-coder-1.3b-instruct performances on HumanEvalExplain (above) and HumanEvalSynthesis (below).

	Pass@1	
Model	**HumanEvalExplain**	**HumanEvalSynthesis**
deepseekcoder-6.7-instruct	0.63	0.80
deepseekcoder-6.7-instruct-8bit	0.60	0.80
deepseekcoder-6.7-instruct-4bit	0.57	0.75
deepseekcoder-1.3-instruct	0.46	0.49
deepseekcoder-1.3-instruct-8bit	0.48	0.47
deepseekcoder-1.3-instruct-4bit	0.41	0.47

Table 4.4: Evaluation of the quantized models against the original model on HumanEval-Explain and HumanEvalSynthesis.

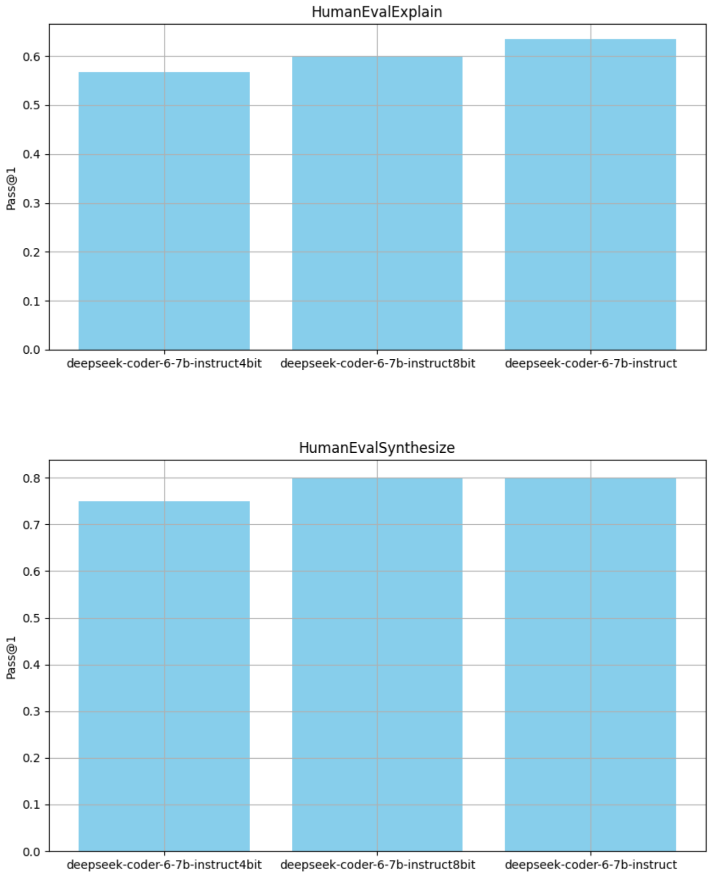


Figure 4.9: Effect of quantization on deepseek-coder-6.7b-instruct performances on HumanEvalExplain (above) and HumanEvalSynthesis (below).

	BLEU (CodeXGLUE version)
deepseek-coder-6.7b-instruct	11.57%
deepseek-coder-1.3b-instruct	12.41%

Table 4.5: Performances evaluated on the CodeXGLUE benchmark.

Prompt:

-*- coding: utf-8 -*-
 	
import calendar
from dateutil.relativedelta import relativedelta
 	
from odoo import fields, models, api,
from odoo.exceptions import UserError
 	
class ResCompany(models.Model):
_inherit = 'res.company'
 	
def _autorise_lock_date_changes(self, vals):
"'Check the lock dates for the current companies. This can't be done in a api.constrains because we need
to perform some comparison between new/old values. This method forces the lock dates to be irreversible.
* You cannot set stricter restrictions on advisors than on users.

Table 4.6: Prompt used to test models fine-tuned on local code.

These results highlight the limitations of relying on the BLEU score. When the model's responses deviate from the ground truth, the BLEU score may fail to reflect their correctness, even if the answers are accurate. Therefore, we stop using this benchmark and continue evaluating code summarization's ability with HumanEvalExplain.

4.2.3 Local Coders

The two models fine-tuned on the local codebase, **deepseek1.3-localcoder** and **deepseek6.7-localcoder**, cannot be evaluated like the others as there is no benchmark to assess code generation performances across HAUERP and Odoo codebases.

We can still rely on a manual evaluation and generate outputs to analyze the models' behavior. Since they were trained on version 15 of Odoo, we can use the Proof of Concept example described earlier (see 3.4.2). We provide the initial part of the code (see table 4.6) and simply change the word "accountants" to "advisors". This allows us to see whether the models have merely memorized the training set. We also compare their generations to those of their non-fine-tuned counterparts to observe any differences in behavior.

Figure 4.2 shows the answers of the 1.3B models and figure 4.11 the 6.7B models.

Neither of the models regenerates the code for version 16, and both produce different

```python
# -*- coding: utf-8 -*-

import calendar
from dateutil.relativedelta import relativedelta

from odoo import fields, models, api, _
from odoo.exceptions import UserError


class ResCompany(models.Model):
    _inherit = 'res.company'

    def _autorise_lock_date_changes(self, vals):
        '''Check the lock dates for the current companies. This can't be done in a api.constrains because we need
        to perform some comparison between new/old values. This method forces the lock dates to be irreversible.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stric
```

```python
# -*- coding: utf-8 -*-

import calendar
from dateutil.relativedelta import relativedelta

from odoo import fields, models, api, _
from odoo.exceptions import UserError


class ResCompany(models.Model):
    _inherit = 'res.company'

    def _autorise_lock_date_changes(self, vals):
        '''Check the lock dates for the current companies. This can't be done in a api.constrains because we need
        to perform some comparison between new/old values. This method forces the lock dates to be irreversible.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on advisors than on users.
        '''
        if 'lock_date' in vals and vals['lock_date']:
            raise UserError(_('You cannot set stricter restrictions on advisors than on users.'))
        return True

    def _autorise_lock_date_changes_on_advisers(self, vals):
        '''Check the lock dates for the current companies. This can't be done in a api.constrains because we need
        to perform some comparison between new/old values. This method forces the lock dates to be irreversible.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on advisors than on users.
        '''
        if 'lock_date' in vals and vals['lock_date']:
            raise UserError(_('You cannot set stricter restrictions on advisors than on users.'))
        return True

    def _autorise_lock_date_changes_on_users(self, vals):
        '''Check the lock dates for the current companies. This can't be done in a api.constrains because we need
        to perform some comparison between new/old values. This method forces the lock dates to be irreversible.
        * You cannot set stricter restrictions on advisors
```

Figure 4.10: Generation of the original model above and the fine-tuned version below.

```python
# -*- coding: utf-8 -*-

import calendar
from dateutil.relativedelta import relativedelta

from odoo import fields, models, api, _
from odoo.exceptions import UserError


class ResCompany(models.Model):
    _inherit = 'res.company'

    def _autorise_lock_date_changes(self, vals):
        '''Check the lock dates for the current companies. This can't be done in a api.constrains because we need
        to perform some comparison between new/old values. This method forces the lock dates to be irreversible.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on users than on advisors.
        * You cannot set stricter restrictions on advisors than on managers.
        * You cannot set stricter restrictions on managers than on superusers.
        '''
        if 'lock_date_start' in vals or 'lock_date_end' in vals:
            if self.lock_date_start and self.lock_date_end:
                if self.lock_date_start > self.lock_date_end:
                    raise UserError(_('The lock date start must be before the lock date end.'))
                if self.lock_date_start > fields.Date.today():
                    raise UserError(_('The lock date start must be before today.'))
                if self.lock_date_end > fields.Date.today() + relativedelta(years=1):
                    raise UserError(_('The lock date end must be before the end of the year.'))

    @api.model
    def create(self, vals):
        self._autorise_lock_date_changes(vals)
        return super(ResCompany, self).create(vals)

    @api.multi
    def write(self, vals):
        self._autorise_lock_date_changes(vals)
        return super(ResCompany, self).write(vals)
```

```python
# -*- coding: utf-8 -*-

import calendar
from dateutil.relativedelta import relativedelta

from odoo import fields, models, api, _
from odoo.exceptions import UserError


class ResCompany(models.Model):
    _inherit = 'res.company'

    def _autorise_lock_date_changes(self, vals):
        '''Check the lock dates for the current companies. This can't be done in a api.constrains because we need
        to perform some comparison between new/old values. This method forces the lock dates to be irreversible.
        * You cannot set stricter restrictions on advisors than on users.
        * You cannot set stricter restrictions on users than on advisors.
        * You cannot set stricter restrictions on advisors than on managers.
        * You cannot set stricter restrictions on managers than on superusers.
        '''
        if 'lock_date_end' in vals and vals['lock_date_end']:
            if self.env.user.has_group('account.group_account_manager'):
                if vals['lock_date_end'] < self.env.user.company_id.lock_date_end:
                    raise UserError(_('You cannot set a lock date end date that is earlier than the current one.'))
            elif self.env.user.has_group('account.group_account_advisor'):
                if vals['lock_date_end'] < self.env.user.company_id.lock_date_end:
                    raise UserError(_('You cannot set a lock date end date that is earlier than the current one.'))
            elif self.env.user.has_group('account.group_account_user'):
                if vals['lock_date_end'] < self.env.user.company_id.lock_date_end:
                    raise UserError(_('You cannot set a lock date end date that is earlier than the current one.'))
        if 'lock_date_start' in vals and vals['lock_date_start']:
            if self.env.user.has_group('account.group_account_
```

Figure 4.11: Generation of the original model above and the fine-tuned version below.

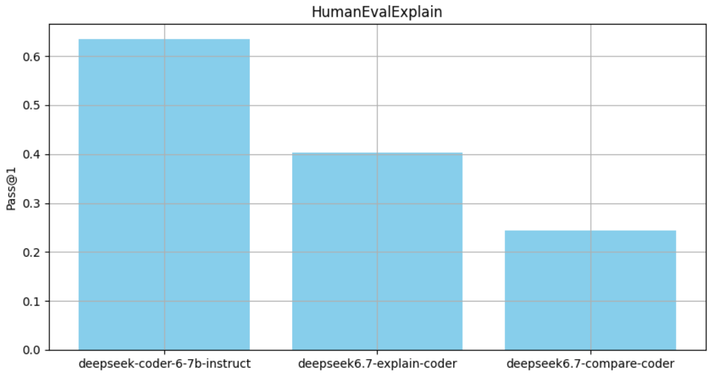


Figure 4.12: Evaluation of the fine-tuned models against the original model on HumanEvalExplain.

completions. As anticipated, fine-tuning clearly affects their output.

For the 1.3B models, fine-tuning appears to have a positive impact, as the model now generates code rather than repeating the same comment.

However, without expertise in the Odoo API, it is challenging to determine whether the results are truly beneficial. Additionally, we cannot state whether one model is better than the other, nor if the 6.7B version outperforms the 1.3B version. To accurately assess their performance, the models need to be deployed and actively tested by developers who can evaluate their accuracy and usefulness, and compare the performances through their expertise.

4.2.4 HumanEvaluation: Code Summarization and Synthesis

Starting at this point, we aim to evaluate the performance of our fine-tuned models and compare them to the original model.

Figure 4.12 shows the performance of **deepseek6.7-explain-coder** and **deepseek6.7-compare-coder** in comparison to **deepseekcoder-6.7-instruct**. We observe a clear degradation in code summarization capabilities.

Since the evaluation on this benchmark is directly linked to the models' code synthesis abilities, we assessed code synthesis performance as well to ensure that the observed degradation in summarization is not a result of diminished synthesis capabilities. Figure 4.13 indicates that there is indeed some degradation in code synthesis. However, it is not significant enough to conclude that this is the primary cause of the decline in summarization performance. To avoid overestimating our models, it is more prudent to conclude that the fine-tuning processes have undermined our synthesis abilities.

The exact pass@1 measurements are reported in 4.7.

4.2.5 Conala Benchmark: Evaluate Code Generation

Fine-tuned models are evaluated on the Conala benchmark and compared to the original model. The smoothed 4-grams BLEU score is measure, and the results are displayed in figure 4.14 and reported in table 4.8.

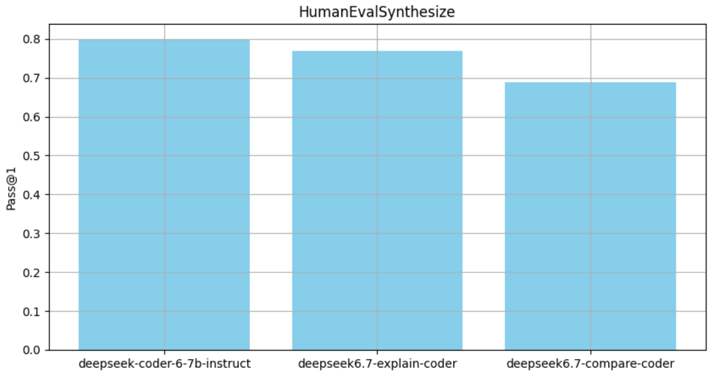


Figure 4.13: Evaluation of the fine-tuned models against the original model on HumanEvalSynthesis.

	Pass@1	
Model	**HumanEvalExplain**	**HumanEvalSynthesis**
deepseekcoder-6.7-instruct	0.63	0.80
deepseek6.7-explain-coder	0.40	0.77
deepseek6.7-compare-coder	0.24	0.68

Table 4.7: Evaluation of the fine-tuned models against the original model on HumanEval-Explain and HumanEvalSynthesis.

Fine-tuned models outperform the original one, but **deepseek6.7-compare-coder** is the one that achieves the highest precision while having been fine-tuned for a different task.

4.2.6 CompareEval: Evaluate Code Comparison

Fine-tuned models are evaluated on the CompareEval benchmark and compared to the original model. The smoothed 4-grams BLEU score is measured, with results displayed in figure 4.15 and reported in Table 4.9.

The two fine-tuned models outperform the original model. Notably, **deepseek6.7-compare-coder** achieves the highest precision by improving its precision by a factor of 206.25%.

	Conala
Model	BLEU4
deepseekcoder-6.7-instruct	0.29
deepseek6.7-explain-coder	0.35
deepseek6.7-compare-coder	0.36

Table 4.8: Evaluation of the original model and the fine-tuned versions on Conala benchmark.

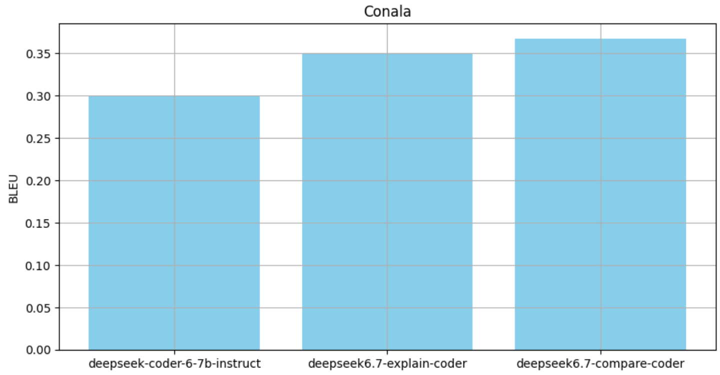


Figure 4.14: Evaluation of the original model and the fine-tuned versions on Conala benchmark.

	CompareEval
Model	BLEU4
deepseekcoder-6.7-instruct	0.0016
deepseek6.7-explain-coder	0.0029
deepseek6.7-compare-coder	0.0049

Table 4.9: Evaluation of the original model and the fine-tuned versions on CompareEval benchmark.

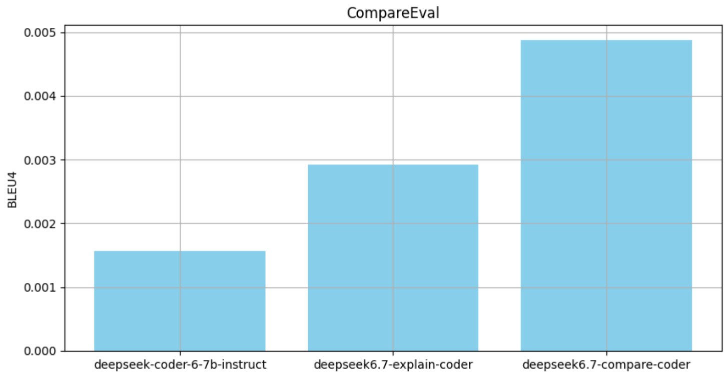


Figure 4.15: Evaluation of the original model and the fine-tuned versions on CompareEval benchmark.

4.3 Final Discussion

Before concluding, we summarize the key observations from these experiments.

DeepSeekCoder's instruction-tuned models exhibit strong performance in code understanding tasks.

Fine-tuning on a single GPU is challenging but feasible within our framework. However, reflecting on the training results is complicated by the use of QLoRA and other simplifications, which may introduce errors. As a result, the high loss observed could be partly due to these factors.

During fine-tuning, while cross-entropy loss may remain high, improvements are still observed on benchmark evaluations. Since different benchmarks use varying metrics, the relationship between cross-entropy reduction and benchmark performance is not straightforward. It remains unclear how much cross-entropy improvement is needed to reach a specific level of benchmark precision, as variability is expected across different metrics.

Significantly, we demonstrate that fine-tuning on the dataset the benchmark originates leads to better performance on that benchmark. Resulting in enhanced abilities for the task evaluated by the benchmark. This remains true even if we fine-tune for a different task than the one we evaluate for, like Conala. We do not speculate on the reasons for this, as it may simply be due to the small size of the dataset, which could cause the model to learn some examples by heart.

The experiments also show a correlation between various coding abilities, though this correlation can be either positive or negative. For instance, fine-tuning on CoNaLa and CommitPackFT improved performance on their respective benchmarks but negatively impacted results on HumanEvalExplain and HumanEvalSynthesis. Also, fine-tuning on CommitPackFT resulted in better performance on the Conala benchmark than fine-tuning directly on the Conala dataset.

The interdependence between HumanEvalExplain and HumanEvalSynthesis benchmarks, limits the scope of evaluation, as improvements in one can constrain the other and potentially lead to underestimated performances.

Additionally, performance on HumanEval-type benchmarks is influenced by the execution environment, since these tests involve running code. Nevertheless, all evaluations were conducted in the same environment, ensuring fairness in comparison.

Benchmark limitations are also evident with BLEU scores, where comparisons to a ground truth can be misleading, as observed with CodeXGLUE. In code-related tasks, there is often more than one correct solution, and deviations from the ground truth can be valid.

Finally, and most significantly, we demonstrate that fine-tuning on a specific benchmark dataset leads to better performance on that benchmark, resulting in enhanced abilities for the corresponding task.

Chapter 5

Conclusions

This research demonstrates the feasibility of deploying a General Coder's Assistant capable of handling a wide range of code-related tasks relevant to our context.

We first demonstrate that instruction-tuned Large Language Models for code can exhibit a diverse range of code understanding abilities, and that their performance is directly correlated with their code generation capabilities.

We achieve fine-tuning under significant financial and computational constraints by leveraging state-of-the-art methods to minimize memory consumption. Our results also show that small datasets can be sufficient for performance improvement.

Our findings indicate that fine-tuning for specific downstream tasks consistently leads to improved accuracy on related benchmarks, even when cross-entropy loss remains high.

Additionally, we demonstrate a correlation between different coding capabilities: enhancing one can boost others.

Finally, we employ a rigorous evaluation framework, selecting relevant benchmarks and metrics to assess model accuracy across each coding task of interest. We also address the limitations of existing benchmarks in assessing model precision and adopt a more robust evaluation approach focused on comparing model performance comprehensively.

To Go Further

This research can be further enhanced in several ways.

First, the methodology is fully reproducible with any model. If a model exhibits particularly interesting capabilities, it can be fine-tuned and evaluated using the same approach we have proposed.

Second, by removing constraints related to computational power and financial resources, full fine-tuning can be performed without limitations on time or memory. This would eliminate the need for simplifications aimed at reducing memory usage, which can introduce errors and inaccuracies. It would also allow for a clearer assessment of whether fine-tuning for a specific task is feasible or not.

Finally, the CompareEval benchmark could be refined to emphasize correctness by adopting an approach similar to HumanEvalExplain, as discussed earlier in this work.

Appendix A

Experiments with ChatGPT

Early in the research, some experiments were runned with ChatGPT to assess if it was capable of code comparison. The experiments showed promising results (see figure A.1 A.2)

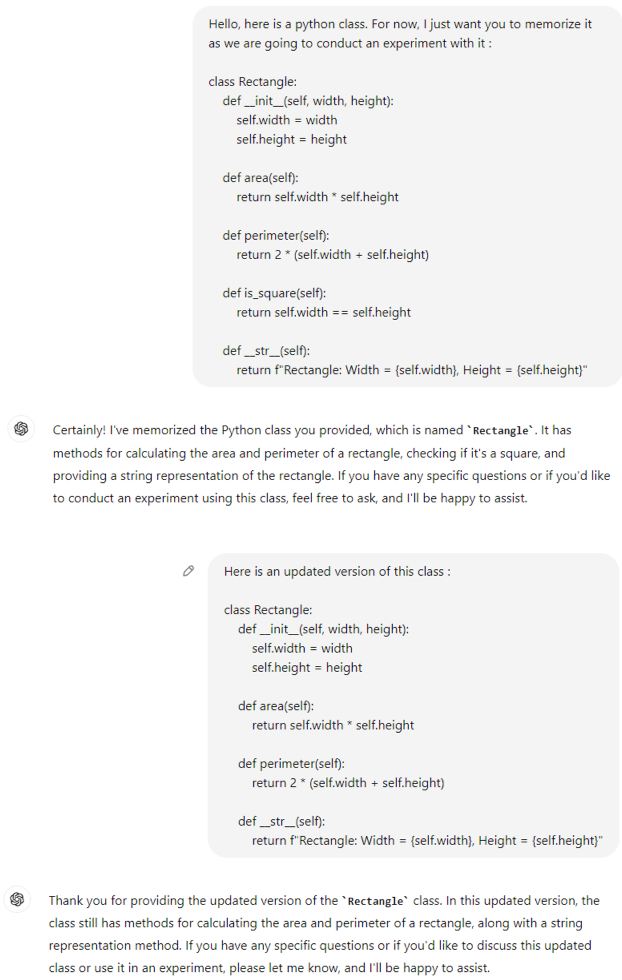


Figure A.1: GPT3 model answers to python code discussions and modifications

Can you describe in natural language all the modifications that were made to obtain this new version ?

In the updated version of the `Rectangle` class, the following modifications were made:

1. The `is_square` method has been removed: The method that used to check whether the rectangle is a square or not has been removed, so this new version does not provide a way to check if the rectangle is a square.
2. The constructor (`__init__`) remains the same: The constructor still takes two arguments, `width` and `height`, to initialize the width and height attributes of the rectangle.
3. The `area` method remains the same: The `area` method still calculates and returns the area of the rectangle, which is the product of its width and height.
4. The `perimeter` method remains the same: The `perimeter` method still calculates and returns the perimeter of the rectangle, which is two times the sum of its width and height.
5. The `__str__` method remains the same: The `__str__` method still provides a string representation of the rectangle, displaying its width and height.

In summary, the modifications made in this updated version involve removing the `is_square` method, but keeping the core functionality of calculating the area, perimeter, and providing a string representation of the rectangle intact.

The consequences of updating your computer program to use the new version of the `Rectangle` class would depend on how your program utilizes the class and whether it relies on the functionality that has been removed in the updated version. Here are some potential consequences:

1. **Loss of `is_square` Functionality**: If your program relied on the `is_square` method to check whether a rectangle is a square, updating to the new class would mean losing this functionality. You would need to find an alternative way to perform this check in your program if it's still necessary.
2. **No Other Major Changes**: Since the constructor, `area`, `perimeter`, and `__str__` methods remain the same, your program's core functionality for creating, calculating area and perimeter, and displaying rectangles would likely remain intact.
3. **Code Compatibility**: If your program uses instances of the `Rectangle` class, you might need to update any code that interacts with these instances to reflect the changes in the class. This includes any code that checks for squareness or relies on the methods that have been removed.
4. **Testing and Debugging**: You should thoroughly test your program with the new class to ensure that the removal of the `is_square` method and any other changes do not introduce unexpected issues or bugs in your application.
5. **Documentation and User Guidance**: If your program is used by others, make sure to update your documentation and user guidance to reflect the changes in the class. This helps users understand the updated functionality.

In summary, updating to the new version of the `Rectangle` class may require some adjustments in your program, particularly if it relied on the `is_square` method. It's important to review and adapt your code as needed to ensure that your program continues to work correctly with the updated class.

Figure A.2: GPT3 model answers to python code discussions and modifications

Appendix B

Failed Fine-Tuning for Code Summarization

The training process repeatedly restarted due to a memory leak issue, despite using an A100-80G GPU with 80GB of memory. We set a high batch size because we were training on a large subset of the original dataset (20,000 samples). However, each time the process completed an epoch and began evaluating the validation set, the memory became saturated.

We had enabled the automatic batch size adjustment option, so instead of crashing, the training resumed from the beginning with half the original batch size. Despite this adjustment, and even when the batch size was reduced to nearly 1, the process still failed during evaluation. We hypothesize that this issue arises from the losses being stored as tensors. As the process attempts to allocate new loss tensors for the validation set, there may not be enough memory available because the training loss tensors consume most of the space.

The lack of documentation on this issue made it difficult to pinpoint the exact cause. Consequently, we decided to abandon the training, especially after also discontinuing the benchmark experiments.

Figure B.1: Failed fine-tuning on CodeXGLUE

Appendix C

Source Code

The source code can be found on this GitHub Repository: Master-Thesis.

Bibliography

[1] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. *Emergent Abilities of Large Language Models*. 2022. arXiv: 2206.07682 [cs.CL] (page 7).

[2] Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. 3rd. Draft Edition. Pearson, 2023. URL: https://web.stanford.edu/~jurafsky/slp3/ (page 8).

[3] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. *Sequence to Sequence Learning with Neural Networks*. 2014. arXiv: 1409.3215 [cs.CL]. URL: https://arxiv.org/abs/1409.3215 (page 8).

[4] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. 2014. arXiv: 1412.3555 [cs.NE]. URL: https://arxiv.org/abs/1412.3555 (page 8).

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL] (pages 8, 9, 12).

[6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. arXiv: 1409.0473 [cs.CL]. URL: https://arxiv.org/abs/1409.0473 (page 8).

[7] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL]. URL: https://arxiv.org/abs/1301.3781 (page 9).

[8] Jeffrey Pennington, Richard Socher, and Christopher Manning. "GloVe: Global Vectors for Word Representation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Ed. by Alessandro Moschitti, Bo Pang, and Walter Daelemans. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. DOI: 10.3115/v1/D14-1162. URL: https://aclanthology.org/D14-1162 (page 9).

[9] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. "Improving language understanding by generative pre-training". In: (2018) (pages 10, 12, 13).

[10] Anthropic. *The Claude 3 Model Family: Opus, Sonnet, Haiku*. 2024. URL: https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf (page 11).

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL]. URL: https://arxiv.org/abs/1810.04805 (page 12).

[12] Kyle Kranen and Vinh Nguyen. *Applying Mixture of Experts in LLM Architectures*. Accessed: 08 05, 2024. 2024. URL: https://developer.nvidia.com/blog/applying-mixture-of-experts-in-llm-architectures/ (page 13).

[13] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. 2021. arXiv: 2005.11401 [cs.CL]. URL: https://arxiv.org/abs/2005.11401 (page 14).

[14] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. *Dense Passage Retrieval for Open-Domain Question Answering*. 2020. arXiv: 2004.04906 [cs.CL]. URL: https://arxiv.org/abs/2004.04906 (page 14).

[15] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. *Large Language Models are Zero-Shot Reasoners*. 2023. arXiv: 2205.11916 [cs.CL]. URL: https://arxiv.org/abs/2205.11916 (page 15).

[16] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL] (pages 15, 17).

[17] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG] (pages 16, 20, 34).

[18] Nvidia. *TensorFloat-32 in the A100 GPU Accelerates AI Training, HPC up to 20x*. Accessed: 08 15, 2024. 2020. URL: https://blogs.nvidia.com/blog/tensorfloat-32-precision-format/ (page 17).

[19] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021. arXiv: 2106.09685 [cs.CL]. URL: https://arxiv.org/abs/2106.09685 (page 18).

[20] Terry Yue Zhuo, Armel Zebaze, Nitchakarn Suppattarachai, Leandro von Werra, Harm de Vries, Qian Liu, and Niklas Muennighoff. *Astraios: Parameter-Efficient Instruction Tuning Code Large Language Models*. 2024. arXiv: 2401.00788 [cs.CL]. URL: https://arxiv.org/abs/2401.00788 (page 18).

[21] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. *CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation*. 2021. arXiv: 2102.04664 [cs.SE]. URL: https://arxiv.org/abs/2102.04664 (page 20).

[22] Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. *OctoPack: Instruction Tuning Code Large Language Models*. 2024. arXiv: 2308.07124 [cs.CL] (pages 20, 21, 25, 28, 34, 36, 37).

[23] Paheli Bhattacharya, Manojit Chakraborty, Kartheek N S N Palepu, Vikas Pandey, Ishan Dindorkar, Rakesh Rajpurohit, and Rishabh Gupta. *Exploring Large Language Models for Code Explanation*. 2023. arXiv: 2310.16673 [cs.SE]. URL: https://arxiv.org/abs/2310.16673 (pages 20, 22).

[24] Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. *WaveCoder: Widespread And Versatile Enhancement For Code Large Language Models By Instruction Tuning*. 2024. arXiv: 2312.14187 [cs.CL]. URL: https://arxiv.org/abs/2312.14187 (page 20).

[25] Balázs Szalontai, Gergő Szalay, Tamás Márton, Anna Sike, Balázs Pintér, and Tibor Gregorics. *Large Language Models for Code Summarization*. 2024. arXiv: 2405.19032 [cs.AI]. URL: https://arxiv.org/abs/2405.19032 (page 20).

[26] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. 2023. arXiv: 2307.09288 [cs.CL]. URL: https://arxiv.org/abs/2307.09288 (page 23).

[27] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. *DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence*. 2024. arXiv: 2401.14196 [cs.SE] (page 26).

[28] Nvidia. *Accelerating a Hugging Face Llama 2 and Llama 3 models with Transformer Engine*. Accessed: 06 05, 2024. 2024. URL: https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/examples/te_llama/tutorial_accelerate_hf_llama_with_te.html (page 27).

[29] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. *RoFormer: Enhanced Transformer with Rotary Position Embedding*. 2023. arXiv: 2104.09864 [cs.CL]. URL: https://arxiv.org/abs/2104.09864 (page 28).

[30] PyTorch. *CrossEntropyLoss*. Accessed: 02 14, 2024. 2023. URL: https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html (page 31).

[31] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. *Training Deep Nets with Sublinear Memory Cost*. 2016. arXiv: 1604.06174 [cs.LG]. URL: https://arxiv.org/abs/1604.06174 (page 33).

[32] Yaroslav Bulatov. *Fitting larger networks into memory*. Accessed: 08 18, 2024. 2018. URL: https://medium.com/tensorflow/fitting-larger-networks-into-memory-583e3c758ff9 (page 34).

[33] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. *CodeBLEU: a Method for Automatic Evaluation of Code Synthesis*. 2020. arXiv: 2009.10297 [cs.SE]. URL: https://arxiv.org/abs/2009.10297 (page 35).

[34] CMU CoNaLa. *CoNaLa: The Code/Natural Language Challenge*. Accessed: 04 20, 2024. 2024. URL: https://conala-corpus.github.io/#dataset-information (page 36).